Filipe Bastos de Freitas

Master of Science

Characterizing and Enforcing Consistency of Online Services

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

Doctor of Philosophy in
Computer Science

Adviser: Rodrigo Seromenho Miragaia Rodrigues,
Full Professor, Lisbon University

Co-adviser: João Carlos Antunes Leitão,
Assistant Professor, Nova University of Lisbon

Examination Committee

Chair: José Augusto Legatheaux Martins
Rapporteurs: Etienne Rivière
João Nuno de Oliveira e Silva
Members: Rodrigo Seromenho Miragaia Rodrigues
José Augusto Legatheaux Martins
José Orlando Pereira
João Manuel dos Santos Lourenço

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

March, 2019

Characterizing and Enforcing Consistency of Online Services

Copyright © Filipe Bastos de Freitas, Faculty of Sciences and Technology, NOVA University Lisbon.
The Faculty of Sciences 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.

This document was created using the (pdf)LaTeX processor, based on the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To Sara, João, and Jaime

Acknowledgements

I would like to thank Professor Rodrigo Rodrigues and Professor João Leitão for their support and guidance on this thesis.

To Professor Nuno Preguiça who was always available to discuss this work, and to Professor João Lourenço and Professor José Orlando Pereira for the comments and suggestion they made to improve the quality of this thesis.

Finally, to my wife Leonor, thanks for the patience you had during this work and sorry for the time that I could not be with you and our children, Sara, João and Jaime.

Abstract

While several proposals for the specification and implementation of various consistency models exist, little is known about what is the consistency currently offered by online services with millions of users. Such knowledge is important, not only because it allows for setting the right expectations and justifying the behavior observed by users, but also because it can be used for improving the process of developing applications that use APIs offered by such services and for creating tools that facilitate this process. To fill this gap, in the first part of this thesis, we present a measurement study of the consistency of the APIs exported by four widely used Internet services, the Facebook Feed, Facebook Groups, Blogger, and Google+. To conduct this study, our work develops a simple, yet generic methodology comprising a small number of tests, which probe these services from a user perspective, and try to uncover consistency anomalies, and reports on the analysis of the data obtained from running these tests for a period of several weeks. Our measurement study shows that some of these services do exhibit consistency anomalies, including some behaviors that may appear counter-intuitive for users, such as the lack of session guarantees for write monotonicity. The results show that developers have to deal with consistency anomalies, to provide consistency guarantees they need.

To address the challenge of enforcing consistency guarantees on top of existing systems, in the second part of this thesis, we show that it is possible to deploy a middleware between the application and the service, which enables a fine-grained control over the session guarantees that comprise the consistency

semantics provided by these APIs, without having to gain access to the implementation of the underlying services. Our solution intercepts all interactions of the client with the online service and uses four different algorithms to enforce each of the session guarantees and also their combination. We evaluated our middleware using the Facebook public API and the Redis data store, and our results show that we are able to provide fine-grained control of the consistency semantics, while incurring in a small local storage and modest latency overhead.

Keywords: Consistency, Distributed Computing, Middleware

Resumo

Existem várias propostas para implementação e especificação de modelos de consistência. No entanto, pouco se sabe sobre a consistência oferecida pelos serviços *online* disponibilizados a milhões de utilizadores. Este conhecimento é importante, não só porque nos permite ter as expectativas corretas sobre o funcionamento do serviço e o comportamento observado pelos utilizadores, mas também porque pode ser usado para melhorar o desenvolvimento de aplicações que usam as APIs fornecidas pelos serviços e ajudar na criação de ferramentas que facilitem este processo. De modo a preencher esta lacuna, na primeira parte desta tese, é apresentado um estudo sobre a consistência fornecida por quatro serviços bastante utilizados na Internet: o Facebook Feed, Facebook Groups, Blogger e Google+. Para conduzir este estudo, foi desenvolvido um método simples e genérico que usa um número pequeno de testes para avaliar violações de garantias de consistências destes serviços segundo a perspetiva do utilizador. O objetivo dos testes é detetar anomalias com base na análise de dados obtidos pela execução de testes durante várias semanas. O estudo mostrou que alguns serviços exibem várias anomalias, nomeadamente comportamentos que podem parecer contraintuitivos aos utilizadores, como a falta de garantias de sessão para escritas monotónicas. Os resultados mostram que os programadores têm de lidar com os vários tipos de anomalias de modo a fornecer as garantias de consistência que necessitam.

Para enfrentar o desafio de garantir consistência aos serviços já existentes, na segunda parte desta tese, mostramos que é possível usar uma camada intermédia entre a aplicação e o serviço, que permite controlo sobre as garantias

de sessão, sem ser necessário ter acesso à implementação do serviço. A solução adotada intercepta todas as interações do cliente com o serviço *online* e usa quatro algoritmos para garantir cada uma das garantias de sessão, bem como a sua combinação. Na avaliação que foi feita usou-se a API publica do Facebook e uma instalação distribuída do serviço de dados Redis. Os resultados mostram que a solução proposta permite garantir controlo sobre as garantias de consistência, com um custo pequeno de armazenamento e um custo modesto em termos de latência.

Palavras-chave: Consistência, Computação Distribuída, *Middleware*

Contents

List of Figures **xvii**

1 Introduction **1**

1.1 Contributions 5

1.2 Publications 5

1.3 Document Organization 5

2 Background and Related Work **7**

2.1 Online Services 7

2.2 Consistency 9

2.3 Replication and Consistency Anomalies 11

2.3.1 Single-Master Replication 11

2.3.2 Multi-Master Replication 15

2.4 Consistency Studies 18

2.5 Middleware solutions for enforcing consistency 20

3 Measurement Study **25**

3.1 Operations 25

3.2 Defining consistency anomalies 26

3.2.1 Session guarantees 26

3.2.2 Divergence 27

3.2.3 Quantitative metrics 28

3.3 Measurement methodology 29

3.3.1 Time synchronization 30

3.3.2 Tests 30

3.4 Results 34

3.4.1 Overall results 37
3.4.2 Session guarantees 38
3.4.3 Divergence 41
3.4.4 Quantitative metrics 44
3.5 Comparison to Related Work 47
3.6 Summary 48

4 Fine-Grained Consistency for Online Services 51
4.1 Target systems 51
4.2 System overview 55
4.2.1 Architecture 55
4.2.2 Overview 56
4.3 Algorithms 57
4.3.1 Read Your Writes 58
4.3.2 Monotonic Reads 61
4.3.3 Monotonic Writes 62
4.3.4 Writes Follow Reads 64
4.3.5 Combining Multiple Session Guarantees 66
4.3.6 Corner Cases 68
4.3.7 Progress 71
4.4 Middleware Design 73
4.5 Evaluation 74
4.5.1 Facebook Results 74
4.5.2 Redis Results 78
4.6 Arguments of Correctness 82
4.6.1 Read Your Writes 82
4.6.2 Monotonic Reads 85
4.6.3 Monotonic Writes 86
4.6.4 Writes Follow Reads 87
4.6.5 Combining Multiple Session Guarantees 88
4.7 Comparison with Related Work 93
4.8 Summary 94

5 Conclusions **97**
5.1 Future work 99

Bibliography **101**

List of Figures

2.1 Service layers 8
2.2 Single-Master Replication 12
2.3 Single-Master Replication, example with one client 13
2.4 Single-Master Replication, example with two client 14
2.5 Multi-Master Replication 15
2.6 Multi-Master Replication, example with one client 16
2.7 Multi-Master Replication, example with two clients 18

3.1 Measurement study method overview 29
3.2 Timeline for Test 1 with three agents. 31
3.3 Timeline for Test 2 with three agents. 32
3.4 Content divergence where computed window is zero 33
3.5 Agents geographical distribution 35
3.6 Percentage of tests with observations of different anomalies 37
3.7 Distribution of Read Your Writes anomalies per test. 38
3.8 Distribution of Monotonic Writes anomalies per test. 39
3.9 Distribution of Monotonic Reads anomalies per test. 41
3.10 Distribution of Writes Follow Reads anomalies per test. 42
3.11 Percentage of tests with content divergence anomalies. 43
3.12 Percentage of tests with order divergence anomalies. 43
3.13 Cumulative distribution of content divergence windows. 44
3.14 Tests where an anomaly of content divergence was observed, but where the window is zero. 45
3.15 Cumulative distribution of order divergence windows. 46

3.16 Tests where an anomaly of order divergence was observed in Google+, but where the window is zero. 46
3.17 Tests where an anomaly of order divergence was observed in Facebook Feed, but where convergence was not reached during the test. 47

4.1 Arquitecture overview 52
4.2 Service get operation, returns N elements of the list 53
4.3 Middleware architecture 55
4.4 Combinations anomaly 68
4.5 Time Gap 72
4.6 Middleware with adapters 73
4.7 Latency of Get Operation in Facebook 75
4.8 Latency of Insert Operation in Facebook 76
4.9 Communication overhead in Facebook 77
4.10 Local storage overhead for Facebook 78
4.11 Latency of Get Operation in Redis 79
4.12 Latency of Insert Operation in Redis 80
4.13 Latency of Insert Operation in Redis in Ireland 81
4.14 Communication overhead in Redis 81
4.15 Local storage overhead for Redis 82

Chapter 1

Introduction

Many computer systems and applications make use of stateful services that run in the cloud, with various types of interfaces mediating the access to these cloud services. For instance, an application may decide to store its persistent state in a Cassandra [27, 46] cluster running on Azure instances, or directly leverage a cloud storage service such as Amazon S3 [9]. At a higher level of abstraction, services such as Twitter [66] or Facebook [36] have not only attracted billions of users to their main websites, but have also enabled a myriad of popular applications that are layered on top of those services by leveraging the public Application Programming Interface (API) they provide [33, 40, 65], for third party application developers.

An important challenge that arises from layering applications on top of cloud APIs (that can either be storage or application-specific APIs) is that the consistency semantics of these cloud services are typically not clearly specified, with studies showing that in practice these services can expose a number of consistency anomalies to applications [50].

This poses several challenges to the programmers who use APIs from such services: it becomes difficult to understand the impact of these consistency semantics on the applications that are layered on top of them; and it forces programmers to modify their code in order to mask or deal with the lack of

desired semantics, which increases development time, complexity, and puts a burden on the programmer of ensuring that the resulting semantics are correct.

In this work, we address this problem through a comprehensive approach that starts by gaining an in-depth understanding of the semantics of online services running in the cloud, and subsequently proposes a set of tools for improving those semantics in a way that facilitates application development. In particular, in the first part of this thesis, we start by systematically understanding what are the consistency guarantees that are effectively offered through the APIs of some of these Internet services. This is important for two main reasons. First, this allows us to better explain and understand the situations where the behavior of the service may be counter-intuitive. Second, this is important to help developers who design applications that interact and make use of these services, to know what they can expect when using these APIs. This allows those programmers to anticipate the effects of using the service on their applications, and to determine if they need to introduce additional logic to mask any undesirable behavior.

For understanding the consistency levels of online service APIs, this work presents the results of a measurement study of the consistency of three popular platforms: Facebook, Google+, and Blogger. Our methodology for conducting this study started by identifying a set of anomalies that are not allowed by several consistency definitions. We then designed two black box tests [44] that probe a given system (through its API) in search of manifestations of these anomalies. We implemented these tests and conducted an extensive consistency measurement experiment in the platforms mentioned above, including two variants of the Facebook API: Facebook Feed and Facebook Group services.

Our study lasted for about a month for each service with a total of 8183 individual tests being executed. Our main findings can be summarized as follows. We found a large prevalence of consistency anomalies in all services with the exception of Blogger. Furthermore, Google+ and Facebook Feed exhibited all anomalies we consider, whereas Facebook Groups exhibited only a subset of them.

Some of these results can be seen as a somewhat natural and even expected

consequence of choosing performance over stronger consistency models such as linearizability. In particular, when the designers of replication protocols choose to provide a fast response to the clients of the service after contacting only one or a small subset of replicas, the implication of this choice is that the consistency model offered by the service has to provide some form of weak consistency, namely one where content divergence is possible, since two writes that were processed by two different replicas may have executed without being aware of each other. We note that our results show an exception to this design principle in the Blogger system, which appears to be offering a form of strong consistency. This can be seen as a sensible design choice considering the write rate and the size of the user base of Blogger.

Overall, one of the most surprising results of our study was to find several types of anomalies that are not inevitable, even when choosing performance over consistency. This is the case of session guarantees, which previous work has shown how to implement, even under weak consistency [64].

Given that some of these services are not providing several session guarantees, according to the results of our study, the relevant question that ensues is what provisions must applications make in order to handle this fact, especially if their application logic could benefit from providing these guarantees. This will imply that developers must enforce session guarantees at the application level, which can be challenging due to the fact developers interact with the service as if it was a black box. Note that some applications may not need to enforce all session guarantees, this choice can be made based on the characteristics of the application. For example, in a social application that targets events, where a single person posts messages about an event, e.g., sports game or a conference, and several people are reading the messages from the feed, it may be enough to return the posts in the order they were issued and that repetitive reads return an increasing number of posts. In this case, it will be necessary to enforce only two session guarantees.

To address this, in the second part of this work, we show that it is possible to build a middleware layer mediating the access to cloud services offering fine-grained control over the consistency semantics exposed to applications,

by enriching the properties originally offered by these services. The idea is to design a library that intercepts every call to the service or storage system running in the cloud, inserting relevant metadata, calling the original API, and transforming the results that are obtained in a transparent way for the application. Through a combination of analyzing this metadata and caching results that have been previously observed, this shim layer can then enforce fine-grained consistency guarantees.

In prior work, Bailis et al. [16] have proposed a similar approach, but with two main limitations compared to this work. First, their shim layer only provides a coarse-grained upgrade from eventual to causal consistency. In contrast, we allow programmers to turn on and off individual session guarantees, where different guarantees have been shown to be useful to different application scenarios [64]. Second, their work assumes the underlying ⟨key,value⟩ store is a NoSQL system with a read/write interface. Such an assumption simplifies the development of the shim layer, since (1) it gives the layer full access to the data stored in the system, and (2) it provides an interface with simple semantics.

Our shim layer allows for a fine-grained control over the session guarantees that applications should perceive when accessing online services. A challenge that arises is that these services typically enforce rate limits for operations issued by client applications. For guaranteeing that this limit is the same when using our shim layer, a single service operation should be executed for each application operation. Furthermore, our layer is not limited to using online storage services with a read/write interface, since it is designed to operate with services that offer a messaging interface such as online social networks. The combination of these three requirements raises interesting challenges from the perspective of the algorithms that our shim layer implements, e.g., to handle the fact that online social networks only return a subset of recent messages, which raises the question of whether a message does not appear because of a lack of a session guarantee or because of being truncated out of the list of recent messages.

We implemented our shim layer and integrated it with the Facebook API and the Redis storage system. Our evaluation shows that our layer allows for

fine-grained consistency upgrades at a modest latency overhead.

1.1 Contributions

The main contributions of this thesis are as follows:

- A generic methodology to probe services and find consistency anomalies;
- A measurement study characterizing the consistency of online services;
- A set of algorithms to provide session guarantees;
- A transparent middleware to provide fine-grained consistency upgrades for online services.

1.2 Publications

The main contributions of this thesis were published in:

Characterizing the Consistency of Online Services (Practical Experience Report). Filipe Freitas, João Leitão, Nuno Preguiça, and Rodrigo Rodrigues. Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN2016)

Fine-Grained Consistency Upgrades for Online Services. Filipe Freitas, João Leitão, Nuno Preguiça, and Rodrigo Rodrigues. Proceedings of the 36th IEEE International Symposium on Reliable Distributed Systems (SRDS2017)

1.3 Document Organization

The remainder of the document is organized as follows. We describe the context of this work and survey related work in Chapter 2. We explain and show the results obtained in our measurement study in Chapter 3. We describe and show the evaluation of our middleware to provide fine-grained consistency in Chapter 4. Finally, conclusions and future work are presented in Chapter 5.

Chapter 2

Background and Related Work

In this chapter, we describe the context of our work and survey the related work that is closest to our consistency measurement study and to our middleware.

2.1 Online Services

Online services are distributed systems used across the world through the Internet. These services are composed of several layers [62], each one with a single responsibility, a service architecture is typically divided into three layers (see Figure 2.1):

Data layer: This layer is responsible to store and retrieve data, and either contains databases such as SQLServer [61] and MySQL [54] or uses cloud data services like Amazon S3 [9].

Processing layer: This layer is responsible to implement the service logic and is composed of application servers that may interact with several internal or external services.

Interface layer: This layer is responsible to provide an interface for clients to operate with the service and is composed of servers, typically HTTP [12], that provide access to Web APIs.

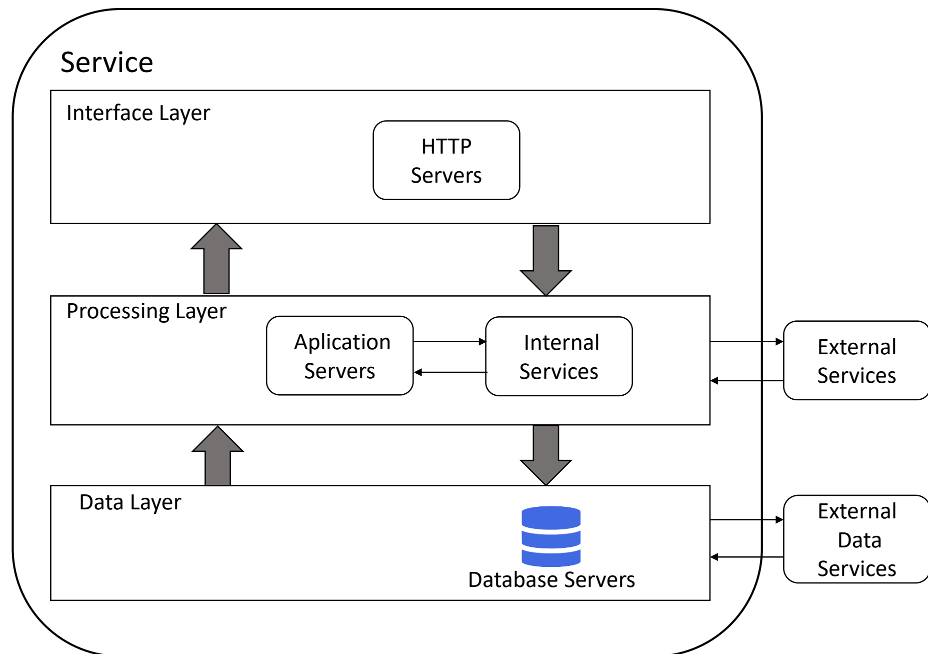


Figure 2.1: Service layers

In order to provide online services to clients across the globe, geo-replication is used both for dependability and good performance (in particular low latency access) [28, 32]. The dependability motivation stems from the ability to tolerate catastrophic failures by having data replicated at multiple sites, whereas the performance gains come from being able to direct users to nearby copies of the data they want to access.

Geo-replication implies replication at the three layers. Applying replication at the interface and processing layers is usually simple because these servers work in a stateless manner. In contrast, replicating at the data layer is complex because it is necessary to replicate the data that comprises the service state, which must be synchronized across replicas.

The price to pay for geo-replication is that the designers of these infrastructures have to deal with an inherent trade-off between performance and consistency, [63]. In particular, if they choose to provide a strongly consistent access to the service, coordination among replicas at different sites becomes a necessity, increasing the latency for request execution. In contrast, if they choose to provide weak consistency, then operations can execute by contacting

a single replica, but the semantics of the service will differ from those of a centralized server (or a strongly consistent system).

2.2 Consistency

Consistency properties provide a way for programmers to know what to expect from services. For instance, a programmer should know if, after making a write, the effects of that write will be reflected in the next operations, or if the effects of a sequence of writes are observed by the same order they where issued. More generally, consistency guarantees are a promise made by the service regarding the observed values by clients given the history of the system (and sometimes the client). Knowing these properties and having services that provide strong consistency helps programmers to reason about what they need to do when writing applications.

Several consistency models exist to describe the consistency of systems, below we present some of these models informally, and we defer a precise definition of the ones we focus on to subsequent chapters.

Linearizability [42] - Ensures that the execution of several operations is equal to some total order that is equivalent with the real-time ordering of the operations, i.e, ensures that the effects of an operation are always reflected before replying.

Causal [47] [3] - Ensures that the execution of several operations respects the happens-before [47] relation between them, whereas non-causally related operations can appear in any order.

Session Guarantees [64] - These various guarantees were defined informally by Terry et al. in their original paper [64] that presented the concept of session guarantees. A session is characterized by a sequence of operations executed during an interval of time. There are four session guarantees:

1. **Read Your Writes** - Ensures that a read operation observes the effects of all writes previously executed in the session.

2. **Monotonic Reads** - Ensures that the effects of all writes observed in a read operation are also reflected in all subsequent reads performed in the same session.

3. **Monotonic Writes** - Ensures that all writes issued in same session are ordered in the order in which they were issued.

4. **Writes Follow Reads** - Ensures that writes are ordered after all writes whose effects have been previously observed in the session making the write.

A system that provides the four guarantees effectively provides the causal consistency model [26]. Since these guarantees are important to this work we explain them in more detail in Section 3.2.1.

Eventual [67] - Ensures that eventually, when there are no new writes, replicas will have the same state.

Linearizability is a strong consistency model for which low latency is impossible to achieve in geo-replicated scenarios [1, 49]. For example, if a write finishes in one location the effects of that write must be immediately reflected in other locations. This implies that the write must wait until the operation finishes across all locations or at a quorum of replicas.

It is also impossible to implement a distributed data system that provides strong consistency, availability, and partition tolerance simultaneously. Again, if we look to Linearizability, where the effects of a write have to be reflected in all locations or at a majority of replicas immediately, the presence of a network partition makes this impossible to achieve [31], as stated by CAP theorem [24, 39]. It has also been shown that the strongest consistency model that can be enforced with network partitions is the causal model [13, 51]. Despite this, there are scenarios where strong consistency is necessary, and several systems implement strong consistency and try to minimize its cost [17, 29, 52].

The alternative is to use weaker consistency models like the causal model, the session guarantees, or eventual consistency. Eventual consistency is easy to implement, since the system only needs to guarantee convergence in the

absence of writes. Causality and some session properties are more complex to implement, because its necessary to track operation dependencies (e.g., using vector clocks [56]) and enforce the happens-before relation. However it is much easier for developers to develop applications on top of a service that ensures these guarantees, than with eventual consistency, thus several systems implement causal consistency [16, 18, 23, 45, 49, 57, 72].

2.3 Replication and Consistency Anomalies

Services need to scale out in order to provide good performance to clients, as described in Section 2.1. They need to be fault-tolerant, be always available, and provide low latency between the client and the service. To this end, they use replication, i.e., they replicate the data in several servers. In this section we present two replications methods widely used in several systems [25, 53, 54, 58, 60] and show how some consistency anomalies can arise. The intention in this section is not to provide an exhaustive list of consistency anomalies that can arise and why, but just to give an introduction on how the consistency anomalies that we focus in this work can emerge.

2.3.1 Single-Master Replication

The first replication method that we present is Single-Master replication [60]. In this method, there are several servers and one is selected as master. The remaining servers are considered slaves replicas (see Figure 2.2). The master executes write and read requests from the clients and propagates the write operations to the slaves. If there is a problem with the master, one of the slaves takes its place. This eliminates having a single point of failure in the system. In order to balance the load between replicas and to improve the latency between the clients and the service, several systems allow slave replicas to execute read requests [11, 58].

Several consistency anomalies can arise when a service uses this replication method. To show this let us assume a simple scenario where:

1. Servers do not fail.

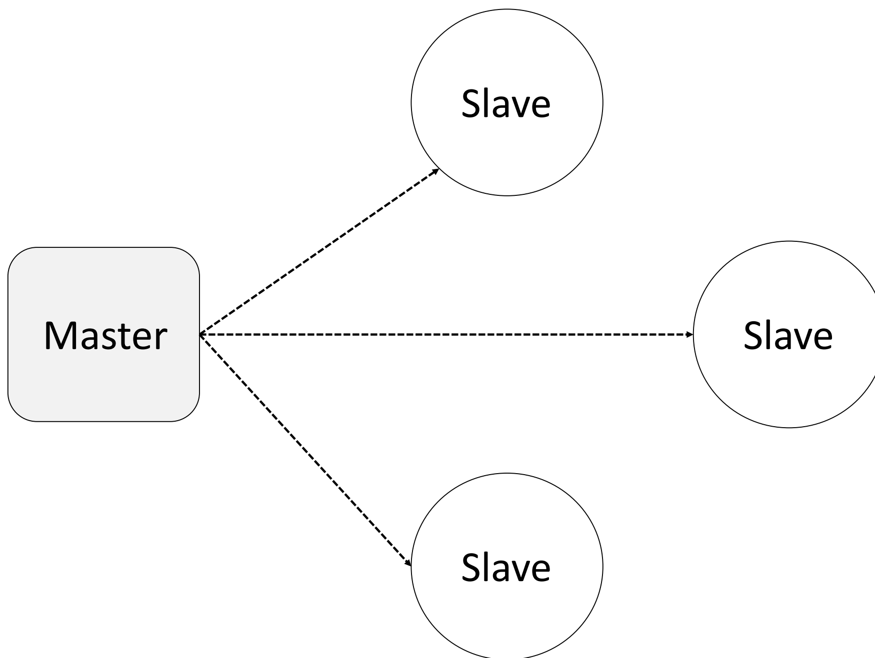


Figure 2.2: Single-Master Replication

2. Write requests terminate when the write operation finished in the master and before being propagated to slaves (asynchronous replication [2]).

3. Writes are guaranteed to be received by the slaves in the order they were issued.

4. Slaves handle client's read requests without synchronizing with other replicas.

The example in Figure 2.3, shows a possible state of a social feed in the different servers after three posts from the same client. The master has the three messages and is propagating them, Slave 1 already has the first two messages and the other slaves only have the first. In this example the following violations of consistency models and properties can occur:

Linearizability - If a client issues a read operation to a slave, it will miss at least one of the messages, and thus the total order that is equivalent with the real-time ordering of the operations is not guaranteed: in this case, the read operation should return a state that reflects the three messages.

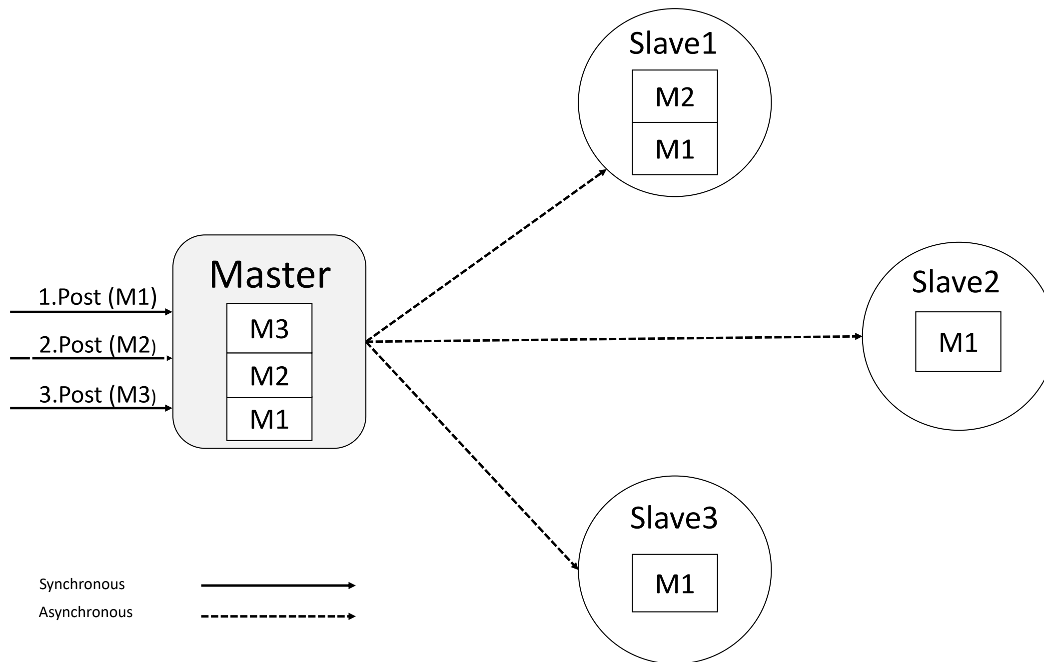


Figure 2.3: Single-Master Replication, example with one client

Session Guarantees - Consider that all write operations are made by client *c* in the same session.

1. **Read Your Writes** - If client *c* issues a read operation to a slave, it may miss at least one of the messages that were posted by *c* previously.

2. **Monotonic Reads** - If client *c* issues a read operation to the master and then to one of the slaves, the second read operations may miss at least one message that was previously observed. This means that successive reads in the same session may not reflect a nondecreasing set of writes.

3. **Monotonic Writes** - This property is guaranteed because we assumed, in this example, that writes are guaranteed to be received by the slave replicas in the order they where issued in the master, which is the order they were issued by client *c* in the session.

4. **Writes Follow Reads** - This property is guaranteed, however we need another example to explain why, Figure 2.4, illustrates a possible state of

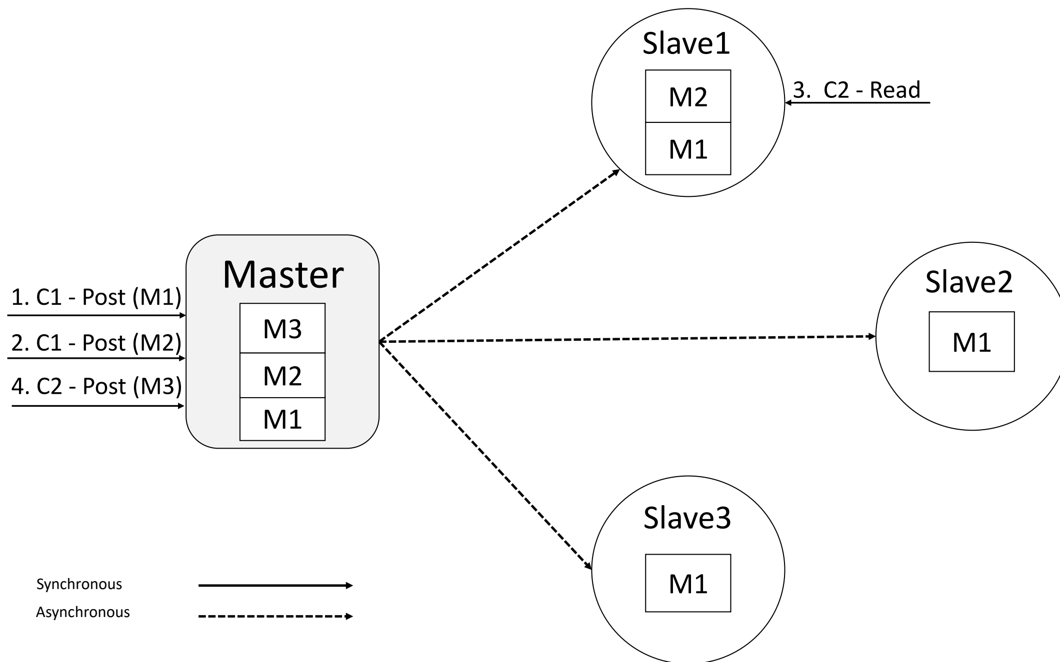


Figure 2.4: Single-Master Replication, example with two client

a social feed in the different servers after two posts from client $c1$ (operations 1 and 2) and a read that precedes a post from client $c2$ (operations 3 and 4), respectively, that returned from Slave 1 the two messages posted by $c1$ (each client executes in the context of its own session). After the execution of all operations, the master received the three messages and is propagating them. Replica Slave 1 has the messages posted by client $c1$ while the remaining slaves replicas only received $M1$. The Writes Follow Reads property is guaranteed because the writes are assumed to be received by the slaves in the order they where issued in the master, which guarantees that if a server returns a message m, the reply must return all other messages issued before m as perceived by the master replica.

Causal - Since the Read Your Writes and Monotonic Reads properties are not guaranteed, causal consistency cannot be guaranteed, a system that provides this consistency model must enforce the four session properties.

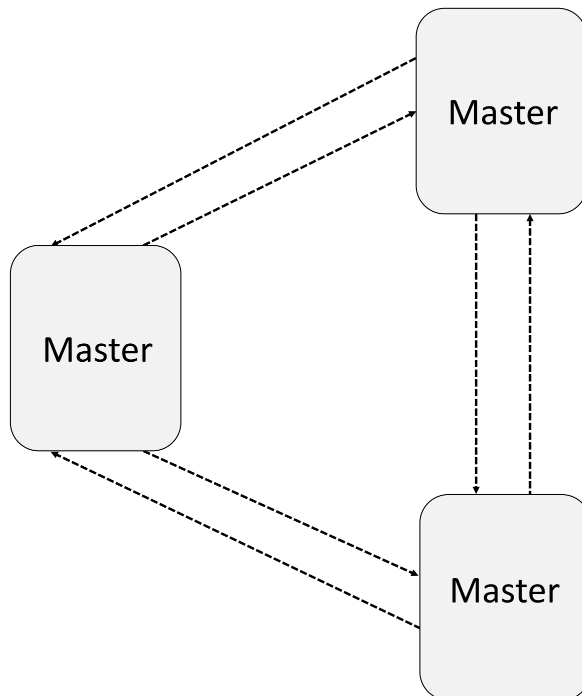


Figure 2.5: Multi-Master Replication

2.3.2 Multi-Master Replication

The second replication method that we present is Multi-Master replication [60]. In this method, each server is a master and can execute writes and reads (see Figure 2.5). This approach avoids having a single point of failure and can balance the load produced by write operations issued by clients among the servers (particularly the cost of replying to clients). However, it can be difficult to implement in systems that need to serialize the writes because there are several masters. In Single-Master replication this can be less complicated since there is only one master and it can be responsible to serialize the writes, and then propagate them in that order.

Several consistency anomalies can arise when a service uses this replication method. To show this, let us assume a simple scenario where:

1. Servers do not fail

2. Write requests terminate when the write operation finished in the replica that received the request from the client, and before being propagated to

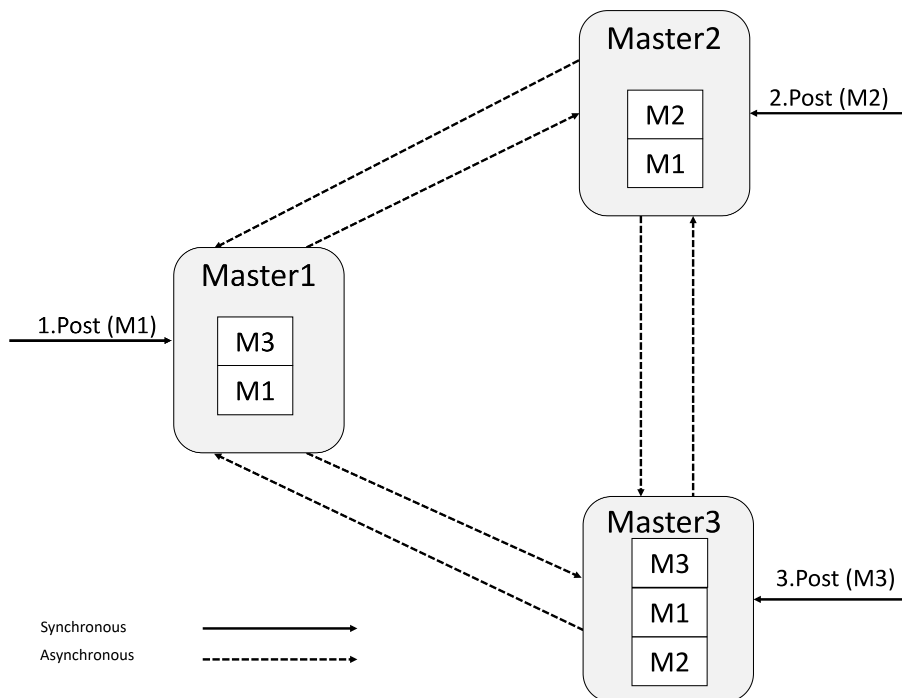


Figure 2.6: Multi-Master Replication, example with one client

other servers (asynchronous replication).

3. Writes originally received from clients by one replica, are guaranteed to be received by all the other servers in the same order.

The example in Figure 2.6 shows a possible state of a social feed in the different servers after three posts from the same client in this case. Each post was made to a different server. After the execution of all operations: Master 1 has messages $M3$ and $M1$ and already propagated $M1$ to all servers; Master 2 has messages $M2$ and $M1$ and is propagating $M2$; Master 3 has all messages and is propagating $M3$. Note that messages $M1$ and $M2$ appear in a different order in Master 1 and Master 2. In this example the following violations of consistency models and consistency properties might happen:

Linearizability - If a client issues a read operation to a server, it will miss at least one of the messages, the total order that is equivalent with the real-time ordering of the operations is not guaranteed: in this case, the read operation should return the three messages.

Session Guarantees - Consider that all write operations are made by client c in the same session.

1. **Read Your Writes** - If client c issues a read operation to Master 1 or Master 2, it may miss one of the messages that were posted by c previously.

2. **Monotonic Reads** - If client c issues a read operation to Master 3 and then to one of the other masters, the second read operation may miss one message that was previously returned. This means that successive reads in the same session do not always reflect a nondecreasing set of writes.

3. **Monotonic Writes** - This property is not guaranteed because in Master 1 message $M2$ is missing and in Master 3, $M2$ and $M1$ are in a order that is not consistent with the order in which the client wrote those messages ($M1$ before $M2$). This occurs because the posts were made in different servers and some messages were delayed. Note that we assumed that writes are guaranteed to be received by all the servers in the order they where issued in the origin master, but this is not enough, because the order between messages posted across different masters may diverge due to propagation delays.

4. **Writes Follow Reads** - This property is not guaranteed, however we need another example to explain why. Figure 2.7 shows a possible state of a social feed in the different servers after two posts from client $c1$ (operations 1 and 2) and a read that precedes a post from client c2 (operations 3 and 4 correspond to the read and the post, respectively). In this example, the read executes at Master 2 and returns the two messages posted by c1. We assume, in this case, that each client is in a different session. After the execution of all operations: Master 1 contains messages $M3$ and $M1$; Master 2 has the messages posted by client $c1$ and is propagating $M2$; Master 3 has the three messages and is propagating $M3$. In this case, the Writes Follow Reads property is not guaranteed because $M2$ is missing in Master 1, but $M3$ was issued after an observation of $M2$, which by this property restricts all servers to only expose $M3$, if they also expose $M2$.

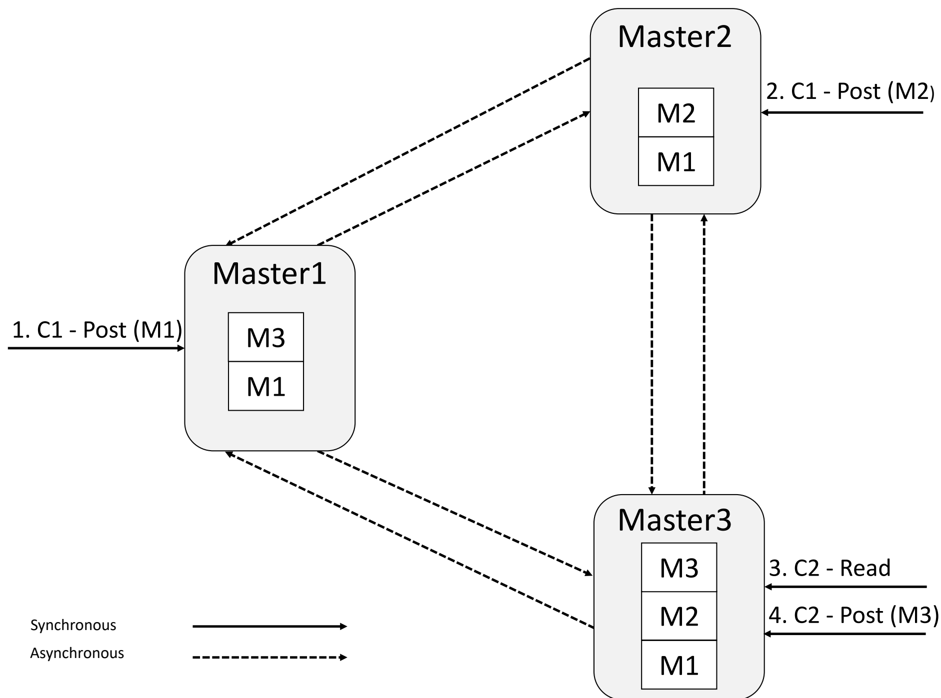


Figure 2.7: Multi-Master Replication, example with two clients

Causal - Since all session guarantees are not guaranteed, causal is not guaranteed, a system that provides this consistency model must enforce the four session properties.

2.4 Consistency Studies

In this section, we present the most relevant related work about consistency measurement studies and their main limitations.

Lu et al. [50] studied the consistency of TAO, Facebook's graph store. The study was performed by logging information inside the infrastructure of Facebook, to build a graph (offline) and detect consistency anomalies. They target linearizability, per-object sequential consistency and the read-after-write anomalies. The methodology used in their work limits the number of services to evaluate, because it needs to log information inside the service. Another limitation is that it may not permit to study consistency as perceived by end users, because some consistency anomalies may be introduced after logging

(e.g., a cache introduced at the interface layer).

Previous authors have conducted studies on the consistency of data storage services. In particular, Wada et al. [68] have focused on testing the properties of Read Your Writes, Monotonic Reads, and Monotonic Writes on several cloud storage services, namely Amazon SimpleDB [7], Amazon S3 [9], Google App Engine Datastore [41], Microsoft Azure Table [14], and Blob Storage. Their study has focused on how the semantics differ depending on the relative location of readers and writers (same or different threads, processes, virtual machines, or geographical regions). To detect consistency anomalies, they perform read and write operations to a data element, in the data storage, and then check for anomalies (e.g., if the client makes a write and in the next read operation the effect of that write is not observed, they detect a read-your-write anomaly). Another relevant study in this context was conducted by Bermbach et al. [19], focusing on the consistency guarantees of Amazon S3 under a heavy load of concurrent writes. These previous studies focus on understanding the consistency properties offered by the storage layer, instead of the consistency provided by the services above this layer. These studies are also using a simple read/write interface to access a data element, but usually services API have support to other operations and to other data structures.

At an even lower layer, Xu et al. [69] conducted a measurement study of response times of virtual machines launched at Amazon EC2. This represents a layer that is even further apart from the one we intent to study. Furthermore, that study focuses only on performance and not on consistency.

Other works have proposed analytic models to determine the consistency properties implemented by distributed key-value stores, based on measurements taken from inside the system. Anderson et al. [10] infer the consistency properties offered by key-value stores through the construction of graphs that capture the operations and their return values, to detect violations of the consistency levels defined by Lamport [48]. Zellag et al. [73] follow a similar approach, building a graph capturing the operations over a system. This graph is enriched with dependencies among object versions for detecting particular consistency anomalies. These studies are focused on key-value stores, and not

in Internet services, which have different interfaces and access restrictions.

Prior work defined a continuous degree of consistency. Bailis et al. [15] model the consistency of weak quorums as a probabilistic bound on staleness, they explain why partial quorums are usually adopted by services and evaluate the staleness of data and the latency. Yu and Vahdat [71] argue that some services can work using weak consistency, but can benefit if the inconsistent interval is bounded. In their proposal they limit the divergence to the final replica state. These proposals share the same core idea with one of the tests introduced further ahead in Chapter 3, that aims at quantifying divergence. However, in contrast to this thesis, the authors of [71] do not verify nor quantify divergence in several Internet services.

2.5 Middleware solutions for enforcing consistency

In this section, we present and discuss the most relevant related work about systems that introduce a layer that can mediate access to a service, in order to upgrade the respective consistency guarantees.

Bailis et al. [16] proposed a system called "bolt-on causal consistency" to provide safety. Their proposal enforces causal consistency by introducing a layer between the client and an eventually consistent data store. This layer has a local storage that contains a consistent causal cut of the data, with all elements requested by the client and their dependencies, and provides two operations: the get operation, to obtain a value given a key and the put operation that receives the value to write and its dependencies. The put operation makes the write in the data store with the associated metadata and updates the local store. The metadata contains a list with the write dependencies and a vector clock (one entry per client). The get operation obtains the value directly from the local storage, tells a background procedure to check if a more recent value for the key is available in the data store, and then returns to the client. The background procedure is responsible for obtaining new values and to guaranteeing that the local store contains a consistent causal cut of the data. This

means that this procedure has to ensure that all write dependencies are stored locally. This approach provides low latency in the read operations, but it may miss the last value for a key. To address this, it is also proposed an alternative read operation, that returns more recent data but may take longer to execute, where the most recent value for a key is obtained from the data store and the necessary dependencies are stored locally, before returning to the client.

There are two main limitations in this previous work: it does not provide a fine-grained choice of which session guarantees (i.e., Monotonic Reads, Monotonic Writes, Writes Follow Reads, and Read Your Writes) the application developer has to ensure so that only the performance penalty that is required for those particular guarantees is incurred. Second, they assume the underlying system offers a general read/write storage interface, which gives significant more flexibility in terms of the system design, in contrast to our work, which is restricted to the APIs provided by Internet services.

Another work that proposes a middleware to guarantee consistency on top of eventually consistent data stores was conducted by Bermbach et al. [20]. This middleware enforces session guarantees and provides a causal behavior to the clients. To enforce the consistency guarantees the middleware relies on vector clocks to track the last value of a key (one entry per application server). The middleware provides operations to write and read the values of a key and caches the values. This work is also limited to a generic storage interface and it does not provide a fine-grained choice to all session guarantees.

In addition to the two previous works, there is also a proposal from Brantner et al. [22] called “Building a Database on S3”, that proposes the design of a middleware to be used on top of a data service (Amazon S3) storing mostly small objects. The middleware provides atomic transactions and can enforce all sessions guarantees. To achieve this, the middleware has a local storage and uses the Simple Queueing System (SQS) [8], the system uses several queues. When a write is committed, the write is sent to the respective queue and a procedure called checkpoint applies the writes to S3. In this system each record is in a page that is stored in S3. To guarantee Monotonic Reads, a commit timestamp is associated to the page. The idea is to use this timestamp

to guarantee that the local storage maintains the most recent versions of the records to return. To guarantee Monotonic Writes the system associates a counter and a client identifier to the pages. The counter increases every time the client commits and the order is enforced in the checkpoint procedure. Read Your Writes is guaranteed if Monotonic Reads is enforced. Writes Follow Reads is also guaranteed because writes are sent to a queue and processed by the checkpoint procedure before being sent to S3. The main limitation of this solution is the overhead of using an external service to enforce the session guarantees, namely, the Simple Queueing System (SQS).

Terry et al. [64] originally proposed the sessions guarantees, that have been referenced above, in the work, "Session guarantees for weakly consistent replicated data". In this work they propose a system that provides the four session guarantees on top of a multi-master replicated data store that provides eventual consistency. The system has a session manager that runs at each client and provides a simple read/write interface. It is assumed that the data store servers are responsible to assign a unique identifier for each new write (WID) and that they must be able to retrieve the set of all WIDs done in each server. For each session, there is a read-set that contains the WIDs of the relevant writes seen in the session and a write-set that contains the WIDs of the writes done in the session. To guarantee Read Your Writes, when a client issues a write, the associated WID is stored in the write-set, when the client issues a read, the session manager needs to find a server that contains the write-set and then read from that server. To guarantee Monotonic Reads, the WIDs of all writes seen in the session are stored in the read-set and, when a client issues a read, the session manager has to find a server that covers the read-set and then reads from that server. To guarantee Monotonic Writes and Writes Follows Reads, it is assumed that the writes observed in a server always precede new writes and that this order is respected while performing replication. Assuming these constraints, to guarantee Monotonic Writes, when a write is issued, the session manager has to find a server that contains the write-set and then execute the write in that server. To guarantee Writes Follow Reads, the WIDs of all relevant writes seen in the session must be stored in the read-set, and, when a write

is issued, the session manager has to find a server that contains the read-set and then execute the write in that server. In order to implement the session guarantees more efficiently, the authors propose to replace the sets of WIDs with vector clocks (one entry per server). This avoids having sets of WIDs that grow indefinitely. This solution provides fine-grained consistency; however, it assumes a set of constraints that are hard to achieve when we are working with online services, where the internal logic of the service cannot be changed (i.e., the service is seen as a black box) [44].

To summarize the main limitations of previous work, (1) some solutions do not provide a fine-grained choice of the session guarantees to developers; (2) they assume that the underlying system is a data store with a generic read/write interface, whereas Web APIs have more complex interfaces (e.g. they associate a list of objects to each key and only return part of that list); (3) the use of external services as part of the system (i.e., the Simple Queueing System); (4) they assume several constraints at the server side, such as knowledge of the internal structure of the data store and direct access to all servers, (5) and they assume the absence of requests rate limits when writing and reading from the data store, in contrast, public Web APIs block applications that exceed the limits imposed by the service.

CHAPTER 3

Measurement Study

The goal of the measurement study presented in this chapter is to characterize experimentally the consistency offered by online service through their public APIs. To this end, we designed two tests that probe the service (through the API) in search of consistency anomalies. A particular challenge in this context is that there are multiple consistency definitions, often using different notations. To address this, we define a number of anomalies that are both precise and intuitive to understand by programmers and users of online services. Note that we are not trying to exhaustively define all anomalies that can occur, nor to prove that these are equivalent to any of the various existing consistency definitions. It is also important to point out that if an anomaly is not observed in our tests, this does not imply that the implementation disallows for its occurrence, since it could have been by chance that it did not surface during the period of experimental testing.

3.1 Operations

In the following description, we consider that users (or clients) of the service issue a set of requests, which can be divided into two categories: (1) *write* requests, which create an event that is inserted into the service state (e.g., posting

a new message), and (2) *read* requests, which return a sequence of events that have been inserted into the state (e.g., reading the current sequence of posts). For simplicity, we assume that read operations return the entire sequence of writes. In practice, this could be generalized to define that a read returns a value that is a function of the sequence of writes according to some service specification, that may not contain the whole sequence, in Chapter 4 we are going to discuss this in more detail.

3.2 Defining consistency anomalies

Based on write and read operation categories, we now define the set of anomalies we consider in this study. We split these into three categories:

3.2.1 Session guarantees

The first set of anomalies corresponds to violations of session guarantees [64]. In these definitions we are considering that each client executes in the context of its own session.

Read Your Writes: This session guarantee requires that a read observes all writes previously executed by the same client. More precisely, say W is the set of write operations made by a client c at a given instant, and S a sequence (of effects) of write operations returned in a subsequent read operation of c, a *Read Your Writes* anomaly happens when:

$$\exists x \in W : x \notin S$$

Monotonic Writes: This requires that writes issued by the same client are observed in the order in which they were issued. More precisely, if W is a sequence of write operations made by client c up to a given instant, and S is a sequence of write operations returned in a read operation by any client, a *Monotonic Writes* anomaly happens when the following property holds, where $W(x) \prec W(y)$ denotes x precedes y in sequence W:

$$\exists x, y \in W : W(x) \prec W(y) \wedge y \in S \wedge (x \notin S \vee S(y) \prec S(x))$$

Monotonic Reads: This session guarantee requires that all writes reflected in a read are also reflected in all subsequent reads performed by the same client. In comparison to Monotonic Writes, this has the subtle difference of requiring that the missing write is first observed (i.e., returned by a previous read) by the client before disappearing. More precisely, a *Monotonic Reads* anomaly happens when a client c issues two read operations that return sequences S_1 and S_2 (in that order) and the following property holds:

$$\exists x \in S_1 : x \notin S_2$$

Writes Follow Reads: This session guarantee requires that the effects of a write observed in a read by a given client always precede the effects of writes that the same client subsequently performs. This precludes the situation where a client reacts to a write issued by itself or some other client (e.g., after reading a question that was posed) by issuing another write (e.g., posts a reply), and subsequently some client observes the second write without observing the first one. More precisely, if S_1 is a sequence returned by a read invoked by client c, w a write performed by c after observing S_1, and S_2 is a sequence returned by a read issued by any client in the system; a *Writes Follows Reads* anomaly happens when:

$$\exists x \in S_1 \wedge \exists w \in S_2 : x \notin S_2$$

Note that although this last anomaly has been used to exemplify causality violations in previous papers [5, 49], any of the previous anomalies represent a different form of a causality violation [64].

3.2.2 Divergence

The next two anomalies refer to divergence between the state that is returned by read operations issued by two independent clients [70].

Content Divergence: A content divergence anomaly captures the case where two clients issue read operations and there are at least two writes such that one client sees one but not the other, and for the other client the opposite is true. More precisely, a content divergence anomaly happens when two reads

issued by clients, c_1 and c_2, return respectively, sequences S_1 and S_2, and the following property holds:

$$\exists x \in S_1 \wedge \exists y \in S_2 : y \notin S_1 \wedge x \notin S_2$$

Any system relying on weak consistency protocols is prone to this anomaly, as this is a consequence of the core property of being able to perform and complete a write operation by contacting a single replica in the system (i.e., without synchronization among replicas).

Order Divergence: The order divergence anomaly refers to writes issued by different clients being seen in distinct orders by different clients. More precisely, an order divergence anomaly happens when two reads issued by two clients, c_1 and c_2, return, respectively, sequences S_1 and S_2, containing a pair of events occurring in a different order at the two sequences:

$$\exists x, y \in S_1, S_2 : S_1(x) \prec S_1(y) \wedge S_2(y) \prec S_2(x)$$

where the notation $S(x) \prec S(y)$ denotes that operation x in state S was ordered before operation y.

3.2.3 Quantitative metrics

The anomalies defined so far are boolean predicates over a trace of the system, i.e., they either occur in an execution or they do not. In addition to the presence or absence of these anomalies, we can determine quantitative aspects of the observed behavior.

Content Divergence Window and Order Divergence Window: When considering the two divergence anomalies, it is also relevant to understand how long it takes for the system to converge back to a single coherent state (assuming it eventually does as prescribed by eventual consistency). As such, we can define the *Content Divergence Window* and *Order Divergence Window* as follows. When a set of clients issue a set of write operations, the divergence window is the amount of time during which the condition that defines the anomaly (either content or order divergence) remains valid, as perceived by the various clients.

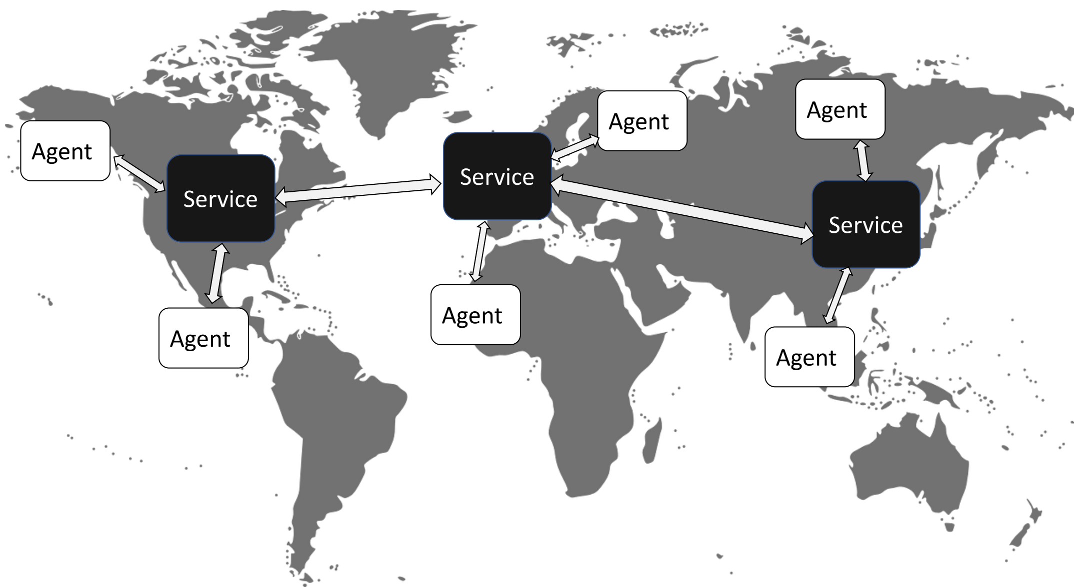


Figure 3.1: Measurement study method overview

3.3 Measurement methodology

In this section, we describe a methodology for testing online services that tries to expose the previously defined anomalies. In a nutshell, our methodology, consists of deploying agents in different points in the globe, as depicted in Figure 3.1. The agents perform several black box tests to the online service, by issuing multiple read and write operations. After the execution of a test, information about the operations is logged locally and then we perform an off-line analysis to detect consistency anomalies.

The notion of a read or a write operation is specific to each service, but should adhere to the specification in Section 3.1. For the services we considered in the realization of this work, since they are either social networking or messaging services, we chose operations that posted a message and listed the current sequence of posts.

3.3.1 Time synchronization

An important aspect of our tests is that they require the clocks of the machines where agents are deployed to be loosely synchronized, for two different reasons. First, we use clock readings to compute divergence windows between different agents. As such, a clock synchronization error could introduce an additional error in the computed values for the divergence windows. Second, some of the tests require the various agents to issue operations as simultaneously as possible (namely to maximize the chances of triggering conditions/executions that maximize the chances of allowing the observation of divergence). As such, a synchronization error would decrease the chances of triggering divergence, and therefore make our measurements of this anomaly more conservative. The synchronization error may reduce the chance of divergence, and thus we expect our result to represent a lower bound on the divergence experienced by these systems.

To synchronize clocks, one could rely on a service such as the Network Time Protocol (NTP) [55]. However, the use of NTP implies releasing the control over the clock synchronization process, which could introduce errors in our measurements when the clock is adjusted. Thus, we disabled NTP and implemented a simple protocol that estimates the time difference between a local clock and a reference clock (which resembles Cristian's algorithm for clock synchronization [30]). In particular, a coordinator process conducts a series of queries to the different agents to request a reading of their current local time, and also measures the RTT to fulfill that query. The clock deltas are then calculated by assuming the time spent to send the request and receive the reply are the same, and taking the average over all the estimates of this delta. The uncertainty of this computation is half of the RTT values.

3.3.2 Tests

Our goal in designing these tests is twofold: first, we want the tests to be complete, in the sense that they allow (and even promote) the possibility of exposing all listed anomalies; and second, we want to make these simple and limit the number of different tests required. Guided by these principles, we

designed the following two tests.

3.3.2.1 Test 1

The sequence of events for the first test is depicted in Figure 3.2. In this test, each agent performs two consecutive writes and continuously issues reads in the background, with a frequency that is determined by the maximum frequency that is allowed by the rate limit of the online service. The writes by the different agents are staggered: agents have sequential identifiers and the first write by agent i is issued when it observes the last write of agent $i-1$. For all operations, we log the time when they occurred (invocation and response times) and their output.

The output of running this test already allows us to detect most of the consistency anomalies from the previous section as follows:

- A violation of Read Your Writes occurs, for instance, when Agent 1 writes M1 (or M2), and in a subsequent read operation M1 (or M2) is missing. (The same applies to each message written by each of the remaining

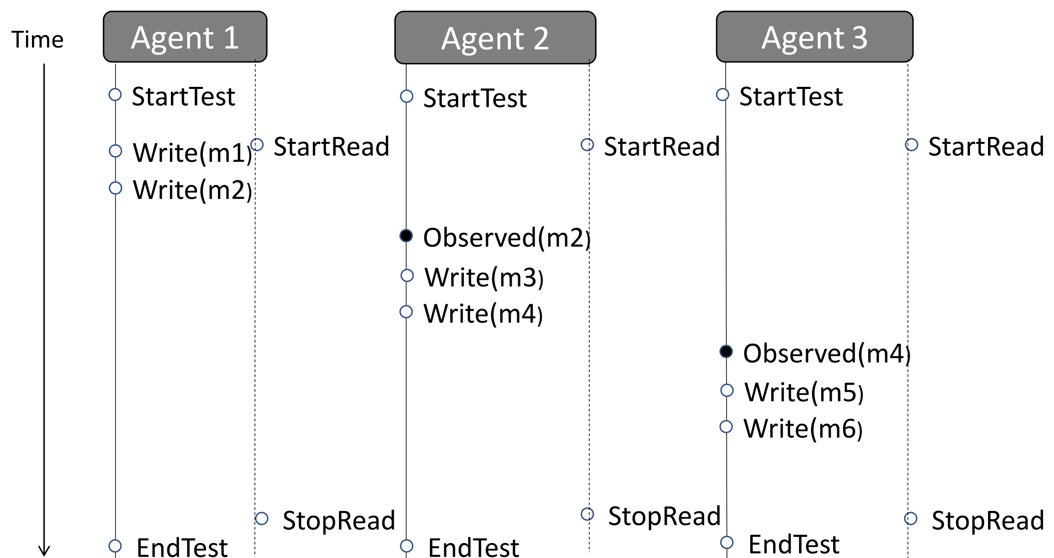


Figure 3.2: Timeline for Test 1 with three agents.

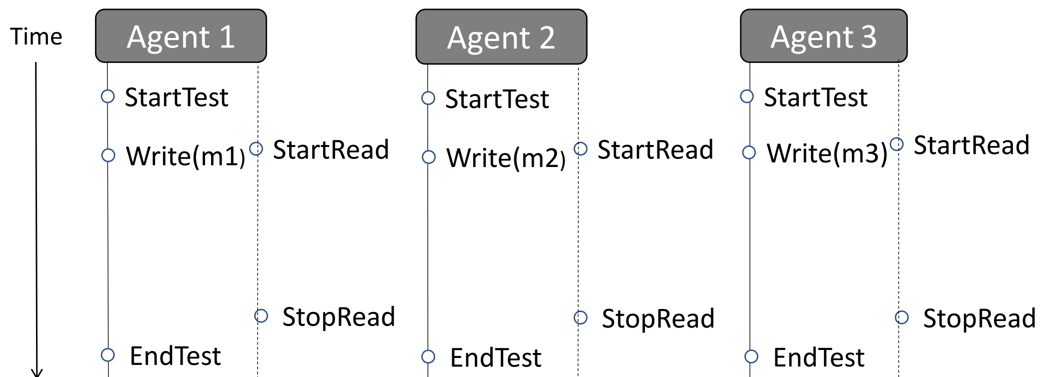


Figure 3.3: Timeline for Test 2 with three agents.

agents.)

- A violation of Monotonic Writes occurs, for instance, when Agent 1 writes M1 and M2, and afterwards that agent either observes only the effects of M2 in the output of a read operation, or observes the effect of both writes in a different order. (The same applies to each pair of messages written by each of the remaining agents.)
- A violation of Monotonic Reads occurs when any agent observes the effect of a message M and in a subsequent read by the same agent the effects of M are no longer observed.
- A violation of Writes Follows Reads occurs when some agent either observes M3 without observing M2 or observes M5 without observing M4. We only consider these particular pairs of messages because, in the design of our test, M3 and M5 are the only write operations that require the observation of M2 and M4, respectively, as a trigger.

3.3.2.2 Test 2

The timeline for the second test is depicted in Figure 3.3. This test attempts to uncover divergence among the view that different agents have of the system, by having all agents issue a single write (roughly) simultaneously, and all agents

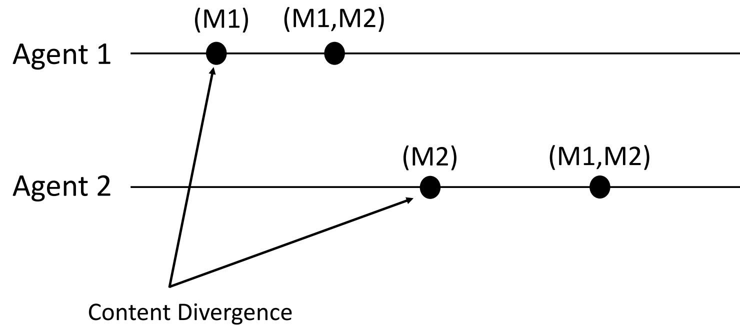


Figure 3.4: Content divergence where computed window is zero

continuously reading the current state in the background. This simultaneity could increase the chances of different writes arriving at different replicas in a different order, and therefore leading the state of such replicas to diverge.

The output of running this test gauges the remaining questions from the previous section. In particular:

- A violation of Content Divergence occurs, for instance, when an Agent observes a sequence of writes containing only M1 and another Agent sees only M2.
- A violation of Order Divergence occurs, for instance, when an Agent sees the sequence (M2,M1) and another Agent sees the sequence (M1,M2).

The content and order divergence windows are also computed using the results of this test by ordering all events according to their absolute time (factoring in the correction for clock deltas as explained previously), and determining the interval of time during which the anomaly conditions hold, as determined by the most recent read. Note that the timeline considering operations from all agents may lead to the following situation, depicted in Figure 3.4: agent 1 reads (M1) at time t_1; agent 1 reads (M1,M2) at t_2; agent 2 reads (M2) at t_3; agent 2 reads (M1,M2) at t_4, with $t_1 < t_2 < t_3 < t_4$. In this case, although there

Agent location	Round-trip time(RTT)
Oregon	136ms
Japan	218ms
Ireland	172ms

Table 3.1: Average RTT between the coordinator and the various agents

has been a content divergence anomaly, the first agent sees the converged state even before the second agent sees the divergence. The conditions we defined apply to the two traces, but there is no instant during the execution where the condition is true about the most recent state seen by the set of agents, leading to a the computed window of zero.

3.4 Results

In this section, we present the results of our measurement study of four online service APIs: Google+, Blogger and two services from Facebook.

For Blogger, we used the API to post blog messages and to obtain the most recent posts [21]. In this service, each agent was a different user, and all agents wrote to a single blog.

For Facebook, we used the Facebook Graph API to interact with both the user news feed [35] and group [34] feed services. In the first case, each user writes and reads from his own feed, which combines writes to the user feed with writes from the feeds of all friends. In the second case, all users are associated with a single group and issue all their write and read operations over that group. In all tests, each agent was a different user, and we used test users, which are accounts that are invisible to real user accounts. We conducted a small number of tests with regular accounts and results were consistent with those obtained using these special test users.

For Google+, we could only find API support for posting “moments”, a moment describes an activity from a user (e.g., a text describing a user activity). We used the API to post a new moment and to read the most recent moments. In this case, all agents shared the same account, since there is no notion of a follower for moments. Unfortunately, there does not seem to exist an extensive

Figure 3.5: Agents geographical distribution

use of the concept of moments, particularly since these are not easy to use through the Web interface. As such, we cannot know whether the results we present apply to what most users observe when they access Google+ services. Note that the behavior of a service accessed through the API might differ from the behavior observed when using the Web Interface.

Subsequently to the experiments, Google+ disabled its support for moments, and Facebook removed the ability to read user news feed from their Graph API.

For each of these services, we ran three agents, which were deployed on three geographically distant locations using different "availability zones" of Amazon EC2, despited in Figure 3.5. In particular, we used the Oregon (US), Tokyo (Japan), and Ireland availability zones, which correspond to different data centers spread across the globe. Furthermore, we deployed an experiment coordinator in another availability zone, in this case North Virginia (US). In Table 3.1 we show the average values we measured for the round trip times between the coordinator and the various agents. These were employed to derive

	Google+	**Blogger**	**FB Feed**	**FB Group**
Period between reads	300ms	300ms	300ms	300ms
Number of reads per agent per test (average)	48	11	14	11
Time between successive tests	34min	20min	5min	5min
Number of tests executed	1036	1028	1020	1027

Table 3.2: Configuration parameters for Test 1

	Google+	**Blogger**	**FB Feed**	**FB Group**
Period between reads	300ms (14X) then 1s	300ms (13X) then 1s	300ms (20X) then 1s	300ms (20X) then 1s
Reads per agent per test	$17-75$	20	40	50
Time between successive tests	17min	10min	5min	5min
Number of executed Tests	922	1012	1012	1126

Table 3.3: Configuration parameters for Test 2

the clock deltas, as discussed in the previous section.

For each of the services, we deployed the various agents for a total period of roughly 30 days per service (for running both tests). For each service, we alternated between running each of the two test types roughly every four days: instances of test 1 ran repeatedly for four days then test 2 for another four days then back to test 1, and so on. Each instance of a test ran until completion: for test 1, the test is complete when all agents see the last message written by Agent3 (M6), and for test 2, the test completes when all agents perform a configurable number of reads. Due to rate limits, after a test instance finishes, we had to wait for a fixed period of time before starting a new one. Before the start of each iteration of a test, the clock deltas were computed again.

Tables 3.2 and 3.3 summarize the parameters we used for configuring each of the tests for each service. These parameters were chosen in a way that minimizes the time to collect the data while taking into consideration rate

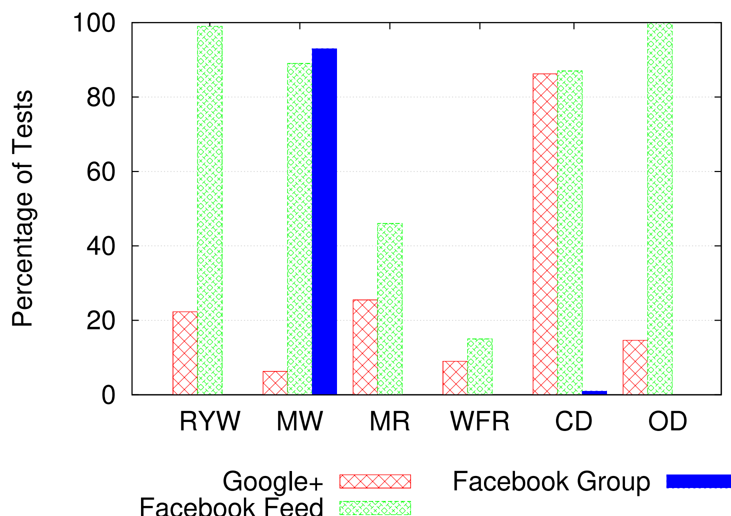
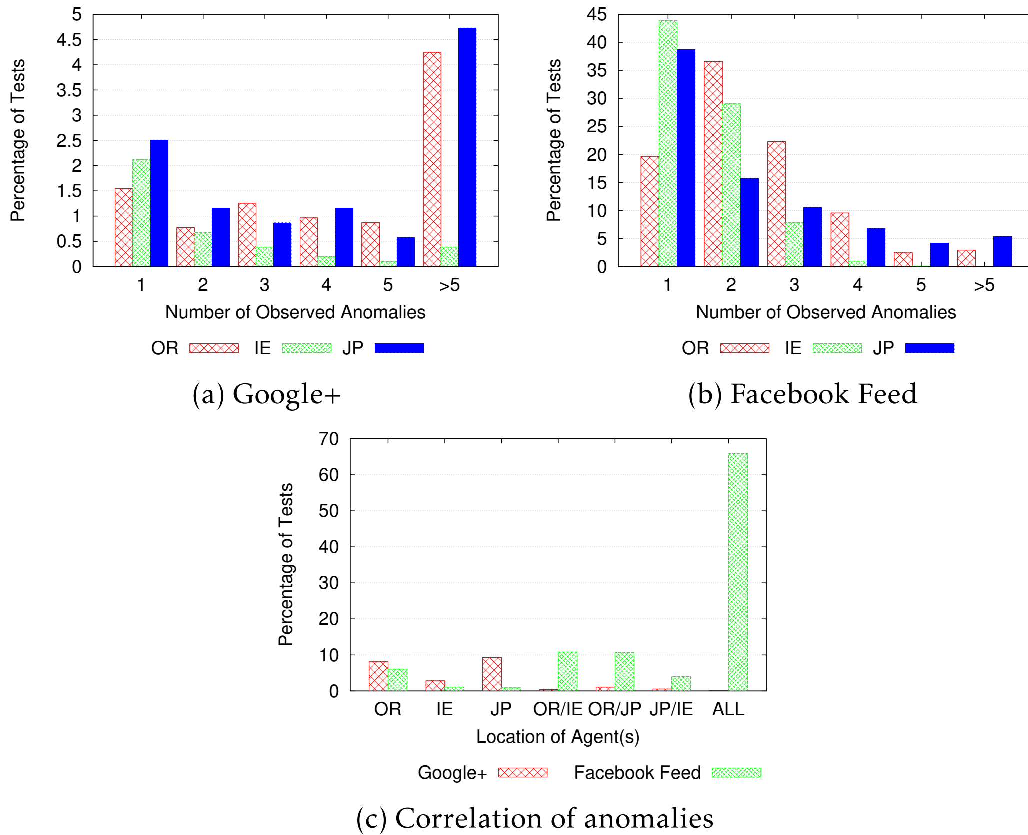


Figure 3.6: Percentage of tests with observations of different anomalies

limits for each service. For the second test, the period between reads is adaptive: initially it is short, and then it becomes one second. This allows for a higher resolution in the period when the writes are more likely to become visible, while respecting the rate limits. In total, we ran 1,958 tests comprising 323,943 reads and 8,982 writes on Google+, 2,040 tests comprising 96,979 reads and 9,204 writes on Blogger, 2,032 tests comprising 195,029 reads and 9,156 writes on Facebook Feed and 2,153 tests comprising 169,299 reads and 9,540 writes on Facebook Group.

3.4.1 Overall results

We start by analyzing the overall prevalence of anomalies in both tests. Figure 3.6 shows, for each anomaly and each service, the percentage of tests where each type of anomaly was observed. All types of anomalies were seen in both Google+ and Facebook Feed, whereas in Facebook Group no violations of Read Your Writes and Order Divergence were observed. For the remaining anomalies, the most common in this service was for Monotonic Writes. In Blogger we did not detect any anomalies of any type. Next, we analyze the occurrence of each anomaly in detail. We omit the results from the pairs of <service, anomaly type> where no anomalies were seen.



(a) Google+

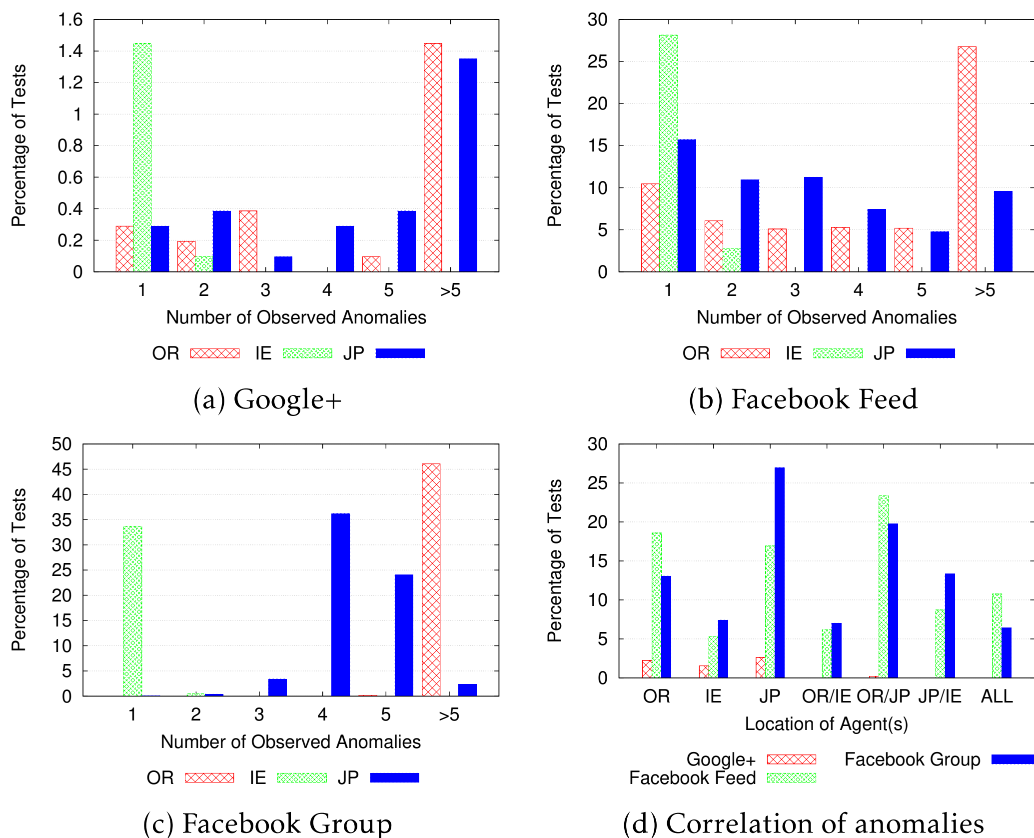
(b) Facebook Feed

(c) Correlation of anomalies

Figure 3.7: Distribution of Read Your Writes anomalies per test.

3.4.2 Session guarantees

We analyze the prevalence of anomalies for each session guarantee. For the **R**ead Your Writes guarantee, Figure 3.6 shows a high value (99%) for Facebook Feed and a visible presence of this type of anomaly (22%) in Google+. Figure 3.7a presents the number of observations of the anomaly per test for Google+. This shows that, in the particular case of Google+, more than half of the tests where this anomaly was detected had several individual violations of the property. The results also show that this anomaly is more prevalent on clients in Oregon and Japan. The results for Facebook Feed, which are reported in Figure 3.7b, show the opposite trend: most occurrences of this anomaly are in tests where it is only detected once or twice per agent. In contrast with Google+, Facebook Feed showed a similar prevalence across client locations. To determine whether these anomalies are correlated across locations, Figure 3.7c depicts the percentage of tests where these anomalies occurred in each agent



(a) Google+ (b) Facebook Feed

(c) Facebook Group (d) Correlation of anomalies

Figure 3.8: Distribution of Monotonic Writes anomalies per test.

exclusively versus across different combinations of the agents. The results show that this does not tend to be a global phenomenon: in Google+, the large majority of occurrences are only perceived by a single agent. However, for Facebook Feed, all three locations perceived the anomaly in a large fraction of tests, because this anomaly arises much more frequently.

Next, we analyze the prevalence of violations of the Monotonic Writes session guarantee, with Figure 3.6 showing a significant prevalence of this type of anomaly both in Facebook Feed and in Facebook Group, with a 89% and 93% prevalence, respectively. Google+ shows a fairly low prevalence with only 6%. The results in Figure 3.8, for Google+, show that this anomaly, when detected in a test, is often observed several times in that test. Additionally, Oregon and Japan have an increased incidence of this anomaly occurring multiple times in a single test, whereas in Ireland, when this anomaly is detected, it often occurs a single time in each test. This phenomenon however might be a consequence of the way that our tests are designed, as in test 1 Ireland is the last client to

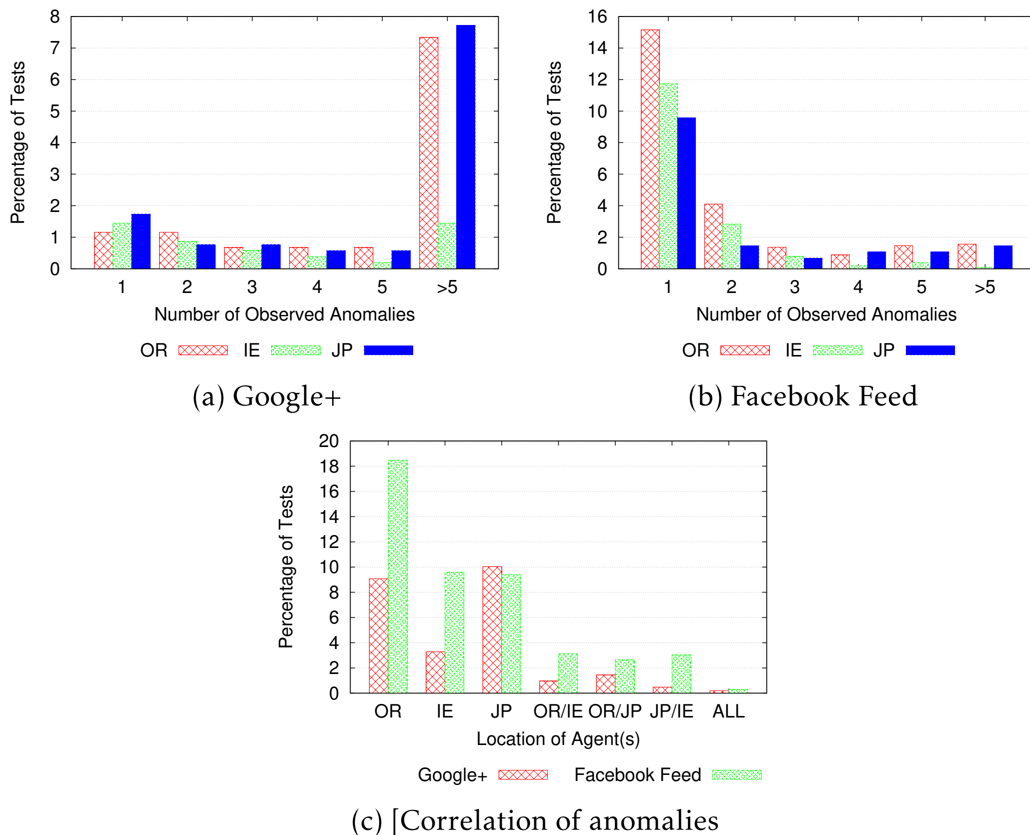
issue its sequence of two write operations, terminating the test as soon as these become visible. Thus, it has a smaller opportunity window for detecting this anomaly. This observation is supported by the fact that the same trend is observed in the results for the Facebook services, and by additional experiments that we have performed, where we rotated the location of each agent.

Figure 3.8d presents the correlation of the location of agents across the tests that observed the anomaly. The figure shows that this tends to be a local occurrence in Google+, where the anomaly is visible in only one of the locations, whereas in Facebook Feed and Group this anomaly tends to be global with a larger prevalence in Japan.

The large occurrence of these anomalies in the Facebook services motivated us to inspect more carefully these phenomena across these services. We noticed that in Facebook Feed, messages are often reordered across different read operations executed by each agent. However, for the particular case of Facebook Group, the reordering of messages occurred mostly in messages issued by the same agent, and that all agents observed this reordering of operations consistently. Upon further inspection, we noticed that each event in Facebook Group is tagged with a timestamp that has a precision of one second, and that whenever two write operations were issued by an agent within that interval (causing them to be tagged with the same timestamp) the effects of those operations would always be observed in reverse order. This suggests that, in this service, this anomaly is produced by a deterministic ordering scheme for breaking ties in the creation timestamp.

The experiment for Monotonic Reads, as shown in Figure 3.6, indicates that 46% of the tests are exhibiting this type of occurrence on Facebook Feed and 25% in Google+. This anomaly was detected in Facebook Group in a single test. Figure 3.9a shows a long tail in the number of observations per test in Google+. Although the anomaly is much more prevalent in Facebook Feed, the results show that it is mostly detected a single time per agent per test. Figure 3.9c indicates a mostly local phenomenon in both services.

Violations of the last session guarantee, Writes Follow Reads, are more frequent in Facebook Feed. As depicted in Figure 3.6, this anomaly only occurs



(a) Google+

(b) Facebook Feed

(c) [Correlation of anomalies

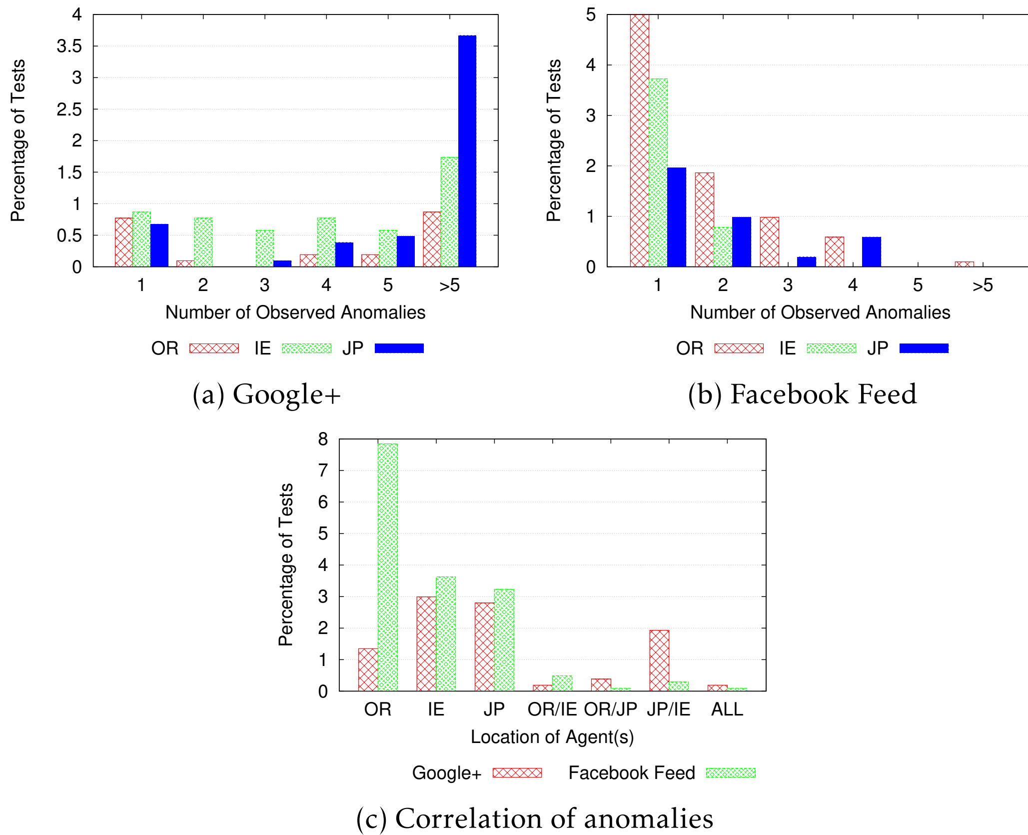
Figure 3.9: Distribution of Monotonic Reads anomalies per test.

in Facebook Group twice. Figure 3.10 shows that, although this is a somewhat frequent anomaly, it does not occur recurrently, with only a few observations occurring per agent in each test for Facebook Feed. In contrast, for Google+ we verify the opposite. Figure 3.10c depicts the set of agents that observe the lack of causality in each test. This indicates a mostly local phenomenon in both services, particularly located in Oregon for Facebook Feed.

3.4.3 Divergence

We now check the presence of divergence events in the collected data. We start by looking at content divergence. Figure 3.11 shows the percentage of tests with divergence between the state observed by pairs of agents.

These results show that content divergence occurs very frequently in Google+ and in Facebook Feed, which indicates the likely use of weakly consistent replication that privileges performance over strong consistency. In particular, the



(a) Google+

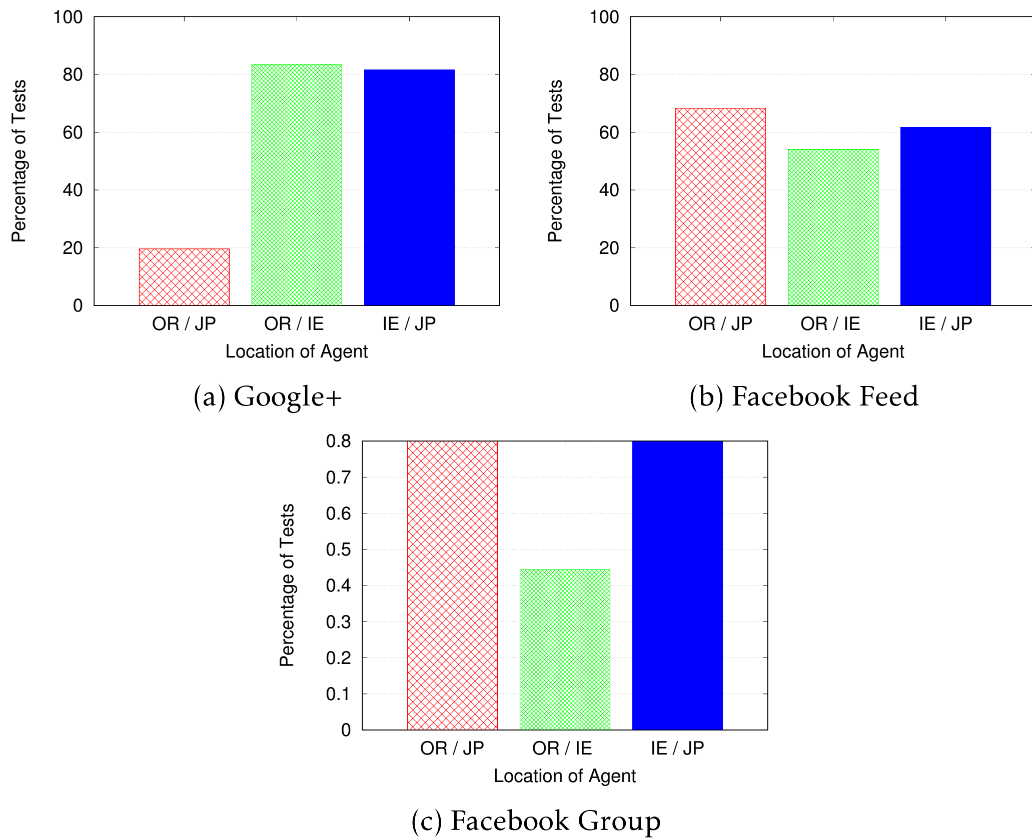
(b) Facebook Feed

(c) Correlation of anomalies

Figure 3.10: Distribution of Writes Follow Reads anomalies per test.

percentage of tests that show content divergence in Google+ is up to 85%, being less pronounced between Oregon and Japan than between the remaining pairs of agents. This might suggest that the Oregon and the Japan agents are connecting to the same data center, whereas the other pairs of agents are not. In the case of Facebook Feed, this occurrence is more uniform across all pairs of agents, and the prevalence is also high (above 50% across all pairs of agents). In the case of Facebook Group the prevalence is extremely low with a total of 15 occurrences of this anomaly, 9 of which happened across a sequence of tests, where the Tokyo agent was unable to observe the operations of other agents. This suggests the agent in Tokyo connects to a different data center than the other agents, hence these anomalies might be caused by a transient fault or network partition.

In terms of the presence of order divergence anomalies, Figure 3.6 shows that this phenomenon occurs in Google+ and in Facebook Feed, with a prevalence that is less pronounced than content divergence in Google+. Similarly to

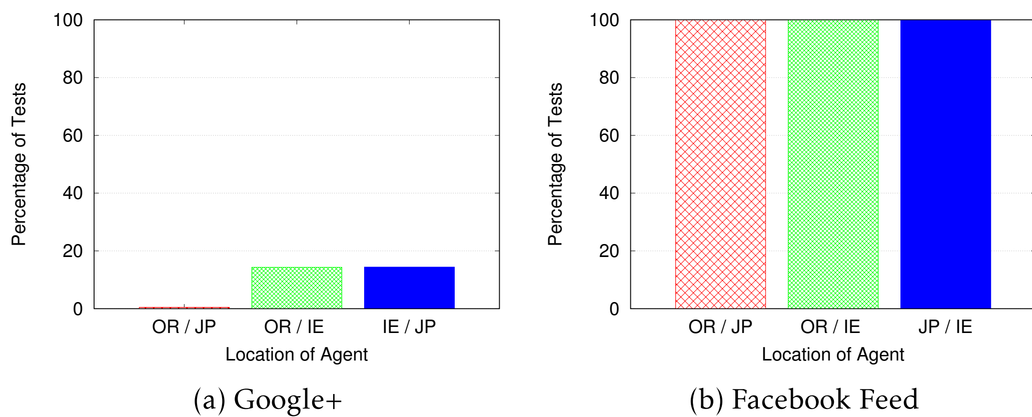


(a) Google+

(b) Facebook Feed

(c) Facebook Group

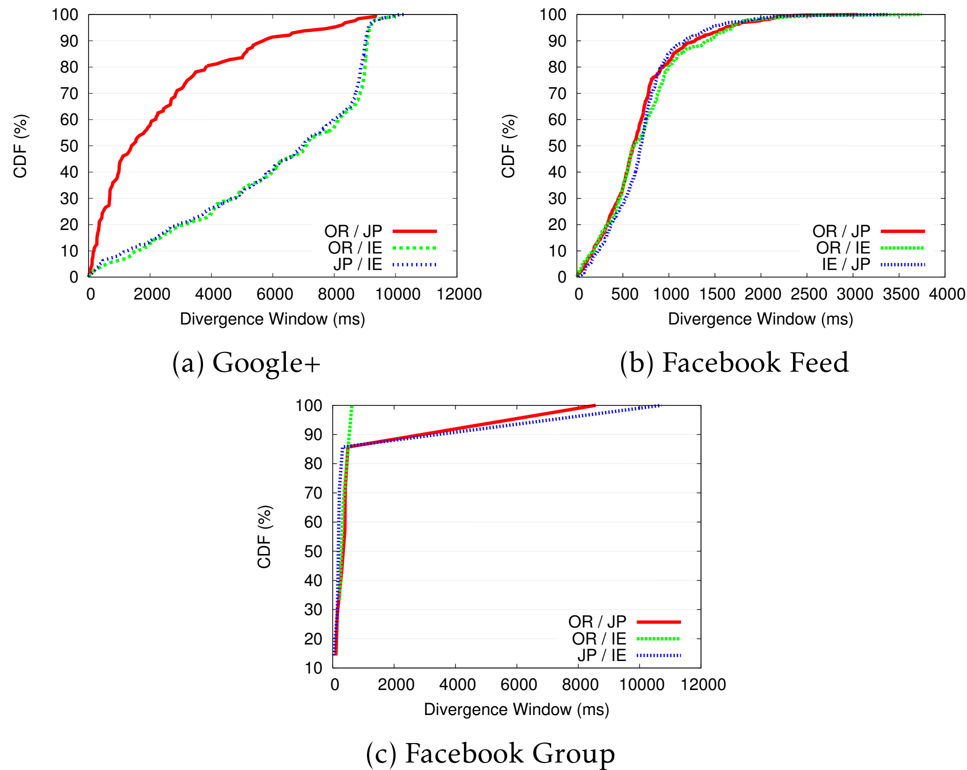
Figure 3.11: Percentage of tests with content divergence anomalies.



(a) Google+

(b) Facebook Feed

Figure 3.12: Percentage of tests with order divergence anomalies.



(a) Google+

(b) Facebook Feed

(c) Facebook Group

Figure 3.13: Cumulative distribution of content divergence windows.

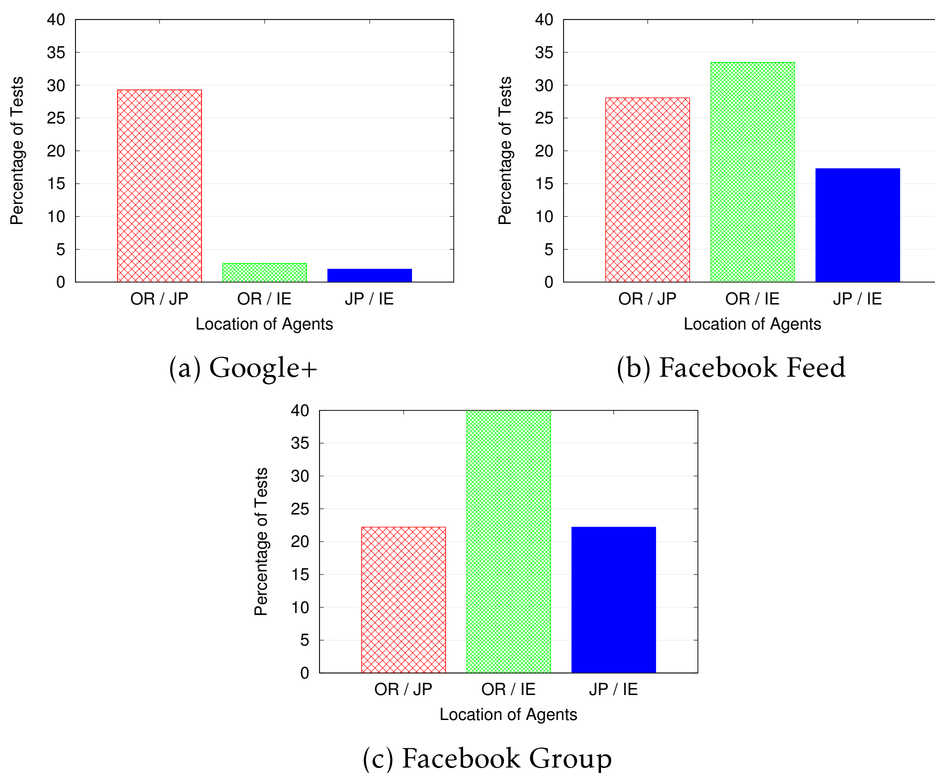
the previously reported results, in Figure 3.12, we observed that this anomaly is much less frequent between Oregon and Japan (below 1%) than between the remaining pairs of agents (around 14%). In Facebook Feed, the prevalence is near 100% across all locations.

While the results for Facebook Feed may seem surprising, they are explained by the semantics of the service. This is because the reply to a read contains a subset of the writes, which are not the most recent ones, but a selection of writes based on a criteria that depends on the expected interest of these writes for the user issuing the read operation.

3.4.4 Quantitative metrics

Next, we analyze several quantitative measurements.

Figure 3.13 shows the CDF of the content divergence window. This distribution is shown for each pair of agents and each service (only considering the largest divergence window for each pair of agents in each test). The results



(a) Google+ (b) Facebook Feed

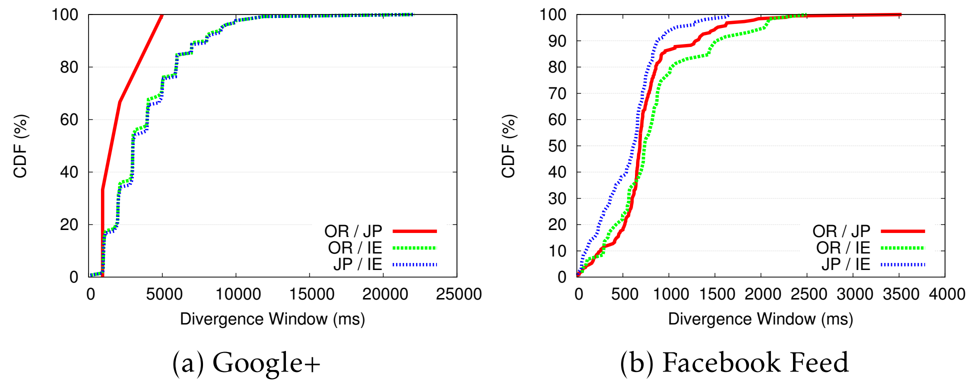
(c) Facebook Group

Figure 3.14: Tests where an anomaly of content divergence was observed, but where the window is zero.

show a smooth distribution of this window, with Google+ taking substantially longer than the remaining services for all agents to regain a consistent view of the system state (on the order of seconds in Google+ versus hundreds of milliseconds in Facebook Feed and most of the content divergence instances observed in Facebook Group). Figure 3.13(a) shows that the Oregon and Japan agents have a convergence time that is much faster than the remaining pairs of agents, and that the other two pairs exhibit similar convergence times. This suggests that writes issued from Oregon and Japan may be processed by the same replica (i.e., data center).

The results depicted in Figure 3.13(b) show that content divergence occurs in Facebook Feed across all agents and all pairs of agents, and takes approximately the same time to converge to a consistent view of the service state (on the order of a few seconds). Again, the semantics of reads in this service may explain these results.

Finally, Figure 3.13(c) depicts the results for Facebook Group, showing that



(a) Google+ (b) Facebook Feed

Figure 3.15: Cumulative distribution of order divergence windows.

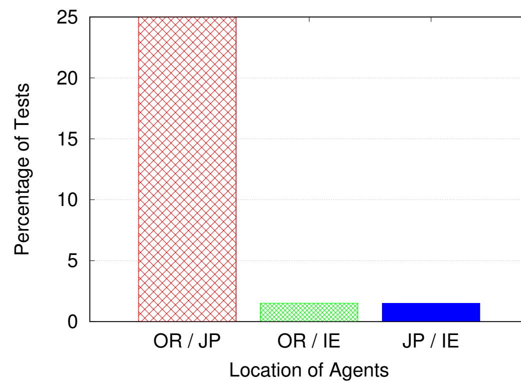


Figure 3.16: Tests where an anomaly of order divergence was observed in Google+, but where the window is zero.

content divergence between the agent in Japan and the two remaining agents takes longer to be resolved. This indicates that the agent in Japan may be contacting a different replica than the remaining agents.

We remind the reader that these CDFs excluded all values where the window was zero, as explained in Section 3.3. In Figure 3.14, we show the percentage of content divergence windows that were computed to have a value of zero according to our methodology. Even though there is a relatively high percentage of more than 30% in Google+ and Facebook Group, we note that this may not be statistically relevant, since there were very few instances of content divergence in this cases. In Facebook Feed the results show a relatively high percentage between all agents that can be associated with the semantics of the service.

The last set of measurements refer to the order divergence window. This

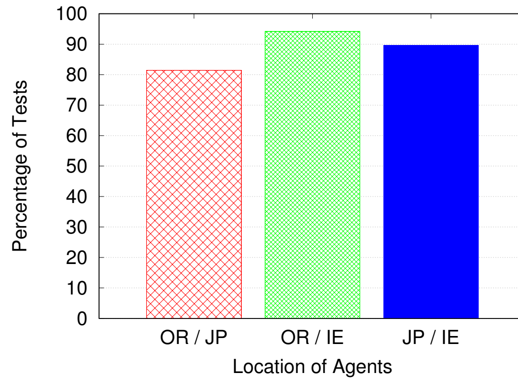


Figure 3.17: Tests where an anomaly of order divergence was observed in Facebook Feed, but where convergence was not reached during the test.

anomaly was only observed in Google+ and Facebook Feed. For Google+, Figure 3.15a shows that a coherent order is more quickly re-established between the Oregon and Japan agents than for the remaining pairs, which can take over ten seconds to achieve this. The reason behind the steps in the curve is that, after the first 12–14 reads, which are more frequent, our agents perform reads with a period of one second, to deal with the imposed limits of the services. As such, the detection of the end of a window is done at the resolution of one second, and in a synchronized way, as shown in the CDF. When looking at the percentage of windows with zero values, reported in Figure 3.16, we note that one of the cases has a high value of 25%, but corresponds to one out of four occurrences. For the Facebook Feed service we observe that a coherent order is established among the several pairs of agents faster.

These results exclude runs where convergence was not reached during the test. In Figure 3.17, we show the fraction of tests where this occurred, between Oregon and Tokyo was 81%, for Oregon and Ireland 94%, for Tokyo and Ireland 89%, again, this might be explained by the semantics of the service.

3.5 Comparison to Related Work

Here, we revisit related work in light of our contributions. In particular, we now focus on a detailed contrast to the more closely related proposals found in the literature.

The most closely related study is the work of Lu et al. [50], which studied the consistency of TAO, Facebook's graph store. The study was performed by logging information inside the infrastructure of Facebook. In contrast, our approach uses the Web APIs of the services, allowing to study the consistency of services, as perceived by end users, without access to their infrastructure. The other main distinction is that our methodology allowed us to study several Internet services, instead of a single one.

The works of Wada et al. [68] and Bermbach et al. [19] have focused on testing the consistency properties on cloud storages, like Amazon S3. In contrast to both studies, our measurement study focuses on understanding the consistency properties offered by service APIs (i.e., above the storage layer), and for the particular case of clients scattered across different geographic locations.

At an even lower layer, Xu et al. [69] conducted a measurement study of response times of virtual machines launched at Amazon EC2. This represents a layer that is even further apart from the one we are analyzing. Furthermore, that study focuses only on performance and not on consistency.

The studies of Anderson et al. [10] and Zellag et al. [73], proposed analytic models to determine the consistency properties implemented by distributed key-value stores, based on measurements taken from inside the system. In contrast, we conduct a measurement study of consistency properties offered by Internet services, focusing on a more general mode than key-value stores.

Finally, the work of Bailis et al. [15], where they modeled the consistency of weak quorums as a probabilistic bound on staleness, and the work of Yu and Vahdat [71] where they limit the divergence to the final replica state, are different from our work, because part of our analysis builds on the idea of quantifying divergence but, in contrast, our goal is to understand its existence in several APIs.

3.6 Summary

We presented a measurement study of the consistency offered by the APIs of four online services. To this end, we started by identifying a set of anomalies

that are not permitted by various consistency levels, and devised two tests that have the ability to expose these anomalies. Our measurement study, based on these tests, ran on Google+, Blogger, Facebook Feed, and Facebook Groups for an aggregate period of one month in each service. Our study showed the relatively frequent occurrence of most of the anomalies across all services except in Blogger.

Chapter 4

Fine-Grained Consistency for Online Services

As we saw in the previous Chapter, developers who design applications for online services such as Facebook or Google+ have little information about their consistency, which exacerbates the complexity of reasoning about the semantics of the application they are developing. In particular, we evaluated a set of services and found a relevant number of consistency anomalies, which motivates the need to automatically enrich the consistency guarantees provided by these services. In this chapter, we are going to present the design of a middleware that provides fine-grained consistency guarantees on top of these services.

4.1 Target systems

Our goal is to provide particular consistency guarantees to third-party applications that leverage popular online web services that expose public APIs. In particular, the application developer may choose to have individual session guarantees (Read Your Writes, Monotonic Reads, Monotonic Writes, and Writes Follow Reads) as well as combinations of these properties (in particular, all four session guarantees when combined corresponds to causality [26]). To achieve this, we provide a middleware that can be easily attached to the third-party

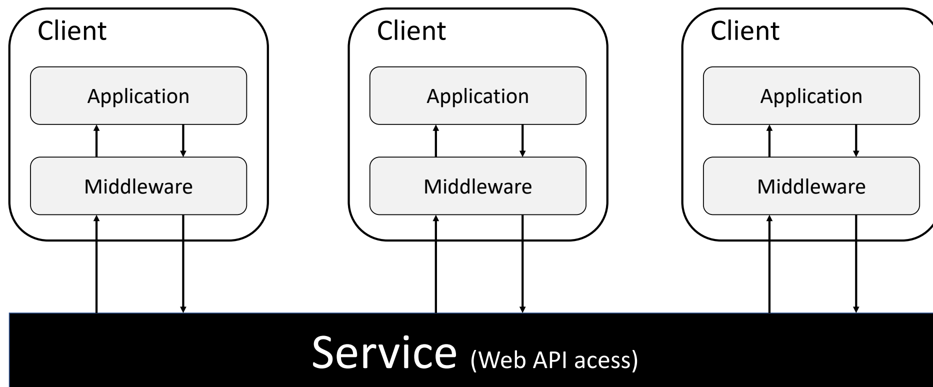


Figure 4.1: Arquitecture overview

client application, allowing us to enrich the semantics exposed through the system public API. Figure 4.1 depicts an overview of our architecture. There are multiple popular systems that provide such public APIs, with various differences in terms of the interface they expose. As such, we needed to focus on a group of APIs with a similar service interface that we can easily adapt to. We chose to focus on a particular class of services, namely social networks, such as Facebook, Twitter, or Instagram. Our choice is based on the relevance and popularity of these services and also on the large number of third-party applications that are developed for them. In particular, we target services that expose a data model based on key-value stores, where data objects can be accessed through a key, and that associate a list of objects to each key. We observe that this data model is prevalent in online social network services, particularly since they share concepts such as user feeds and comment lists. In particular, we target services where the API provides two fundamental operations to manipulate the list of objects associated with a given key: an insert operation to append a new object to the first position of the list, and a get operation, depicted in Figure 4.2, that exposes the first N elements of the list (i.e., the most recent N elements added to that list). Note that most of existing services

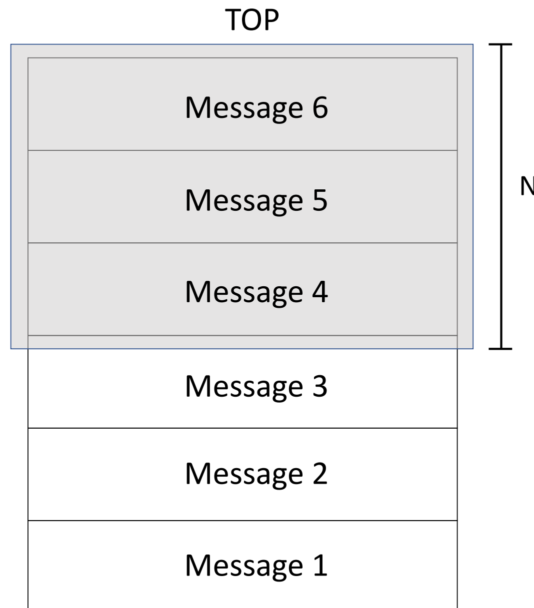


Figure 4.2: Service get operation, returns N elements of the list

specify a maximum of N elements, which means that our middleware must assume that the service can return less than N elements for read operations.

Since we access these services through their public APIs, we need to view the service implementation as a black box, meaning that no assumptions are made regarding their internal operation. Furthermore, we design our protocols without making any assumption regarding the consistency guarantees provided through the public service API. The importance of not assuming any guarantees from existing services is justified by our own previous measurement study, which showed a high prevalence of violations of multiple session guarantees in public APIs provided by services of this class.

Our algorithms require storing metadata alongside the data, which can be difficult to do when accessing services as black boxes, namely when the service has no support for including user managed metadata (this is the case of Facebook, which we explore in the context of our prototype experimental evaluation). In this case, we need to encode this metadata as part of the data itself. As a consequence, when the service is accessed by native clients (i.e., web applications or third party applications that do not resort to our Middleware) the user might observe this metadata. However, we believe that this is not a

crucial issue, since many third party applications only access data objects (i.e, lists) that are used exclusively by that application.

In order to arbitrate an ordering among operations issued by the local client and other remote clients, our Middleware needs an approximate estimate of the current time. To achieve this, two options are available. If the service has a specific call in its public API that exposes the time in the server, such call can directly be used by our system. Otherwise, if the service exposes a Representational State Transfer (REST) API (which is typical in many services) a simple REST call can be performed to the service, and the server time can be extracted from a standard HTTP response header (called *Date*). Note that, even though it is desirable that this estimate is synchronized across clients, we do not require either clock or clock rate synchronization for correctness. In particular, the only negative effect of clocks being out of synch is a reordering of concurrent events from different sessions that is incoherent with their real time occurrence; this can imply, in the case of a service that outputs a sliding window of recent events, that more recent messages may be considered eligible for being truncated (i.e., considered older than the lower end of the window). However, we guarantee that such ordering never violates the correctness conditions we are enforcing.

Finally, we observe that, in practice, the public API exposed by these services often imposes rate limits for operations issued by client applications. These rate limits are exposed under the form of a maximum number of operations that can be executed within a given time window. In particular, we have experimentally observed that violating these rate limits can lead the service to either block further access by the application, or introduce noticeable delays in processing requests issued by the application. The existence of operation rate limits imposes a requirement on our protocols: for each application operation, a single service operation can be issued. This is important to guarantee that an application using our middleware faces the same rate limits as an application using the service directly.

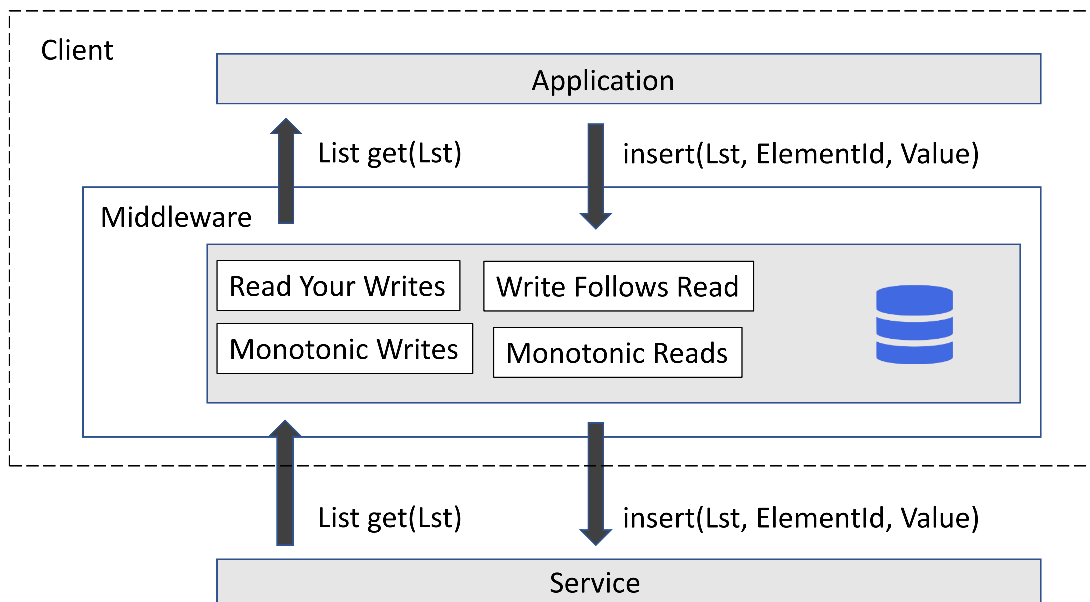


Figure 4.3: Middleware architecture

4.2 System overview

In this section we discuss the general architecture of our solution, which is materialized in a library implementing a middleware layer. We then provide an overview of the operation of our protocols, explaining how they enforce the consistency guarantees of session properties in a transparent way for the client applications.

4.2.1 Architecture

Our system consists of a thin layer that runs on the client side and intercepts every call made by the third-party client application on the service, mediating access to the service. In particular, our layer is responsible for contacting the service on behalf of the client application, process the responses returned by the service and generate responses to the client applications with the session guarantees being enforced. Figure 4.3 provides a simple representation of this architecture.

Our solution can be configured by the third-party application developer

to enforce any combination of the individual session guarantees, defined in Section 3.2.1, namely: *i*) Read Your Writes, *ii*) Monotonic Reads, *iii*) Monotonic Writes, and *iv*) Writes Follow Reads. In order to enforce these guarantees, our system is required to maintain information regarding previous operations executed by the client application, namely previous writes that were issued or previous values that were observed by the client. In addition, our layer can also insert metadata that is stored alongside the data in the original system, but stripped by the library before the final response is conveyed to the client.

4.2.2 Overview

As mentioned above, our system intercepts each request performed by the client application, executes the request in the service, and then processes the answer generated by the service to provide a (potentially different) answer to the client application. This answer is computed based on a combination of the internal state that records the previous operations that were executed by that particular client, and the actual response that was returned by the service.

Tracking application activity. In order to keep track of user activity, our system maintains in memory a set of data structures for each part of the service state that is accessed by the application. These data structures are updated according to the activity of the application (i.e., the operations that were invoked) and the state that is returned by the service. These data structures are: *i*) the *insertSet*, which stores the elements inserted by the client and *ii*) the *localView*, which stores the elements returned to the client.

Enforcing session guarantees. Enforcing session guarantees entails achieving two complementary aspects. First, and depending on the session guarantees being enforced, some additional metadata must be added when inserting operations. As mentioned, this metadata can be either added to a specialized metadata field (if the API exposed by the service allows this) or directly encoded within the body of the element being added to the list. Such metadata has to be extracted by our library when retrieving the elements of a list, thus ensuring transparency towards client applications. Second, our system might be required to either remove or add elements to the list that is returned by the

service when the application issues an operation to obtain the current service state, in order to ensure that the intended session guarantees are not violated. In the next section, we discuss the concrete algorithms executed by our system upon receiving an insert or get operation for a particular list, in order to ensure that the values observed by the client application adhere to the semantics of each of the session guarantees that are intended to be provided.

4.3 Algorithms

We now discuss in more detail the algorithms that are employed by our Middleware layer to enforce session guarantees, and the rationale for their design. Throughout the presentation, we briefly remind what each of the four session guarantees entails and we extend the definitions previously presented in Section 3.2.1 (maintaining the same semantic) to guarantee safety in Web APIs that have the restrictions mention previously. Then we explain why our algorithms ensure that the anomalies associated with each of the session guarantees are prevented by it.

We explain our algorithms assuming that the service offers an interface with the following two functions, which are in practice easily mapped to functions that are supported by the various services that we analyzed: the insertion of an element in a given list *Lst*, denoted by the execution of function `insert`(*Lst*,*ElementID*,*Value*), where *Lst* identifies the list being accessed, *ElementID* denotes the identifier of the element being added (which can be an identifier generated by the centralized service or a unique identifier generated by our Middleware), and *Value* stands for the value of the element being added to the list; and the access to the contents of a list, denoted by the execution of function `get`(*Lst*), where *Lst* identifies the list being read by the client.

When the client accesses a list *Lst* for the first time, a special initialization procedure is triggered internally by our Middleware (Algorithm. 1), which initializes the local state regarding the accesses to *Lst*. The initialization is straightforward: it creates the object *lstState* that maintains all relevant information to manage the accesses to *Lst* (line 2). This state is composed by the

Algorithm 1: Initialization of local state

```
1: upon init(Lst) do
2:    lstState ← init()
3:    lstState.insertSet ← {}
4:    lstState.localView ← {}
5:    lstState.lastTimestamp ← 0
6:    lstState.lastSessionTimestamp ← 0
7:    lstState.insertCounter ← 0
8:    lstState.lastInsertTimestamp ← 0
9:    listStates[Lst] ← lstState
```

sets **insertSet** and **localView** that were discussed previously in 4.2.2, and that are initially empty (lines 3 – 4). Furthermore, four other variables are initialized, **lastTimestamp** and **lastSessionTimestamp**, which are used to maintain information regarding elements that were removed from the **insertSet** and the **localView**, **insertCounter**, which tracks the number of inserts performed by the local client in the context of the current session, and **lastInsertTimestamp** that has the timestamp of the last inserted element in the session. All these variables have an initial value of zero (lines 5 – 8). Finally, the *lstState* variable is stored in a local map, associated to the list *Lst* (line 9). Next, we explain how this local state is leveraged by our algorithms to enforce the various session guarantees. In order to easily combine the algorithms, we divided the get operation in three execution blocks: the read block, that is responsible for obtaining the local state of the list and the data from the service; the transformation block, that is responsible for applying the necessary transformations over the list returned by the service, and the store block, that is responsible for updating the local state.

4.3.1 Read Your Writes

As mentioned in Chapter 3, the Read Your Writes (RYW) session guarantee requires that, in a session, any read observes all writes previously executed by the same client. More precisely, for every set of insert operations W made by a client c over a list L in a given session, and set S of elements from list L returned by a subsequent get operation of c over L, we say that RYW is violated if $\exists\, x \in W : x \notin S$.

Algorithm 2: Read Your Writes

```
1:  function insert(Lst, ElementId, Value) do
2:     lstState ← listStates[Lst]
3:     Element e ← (ElementId, Value)
4:     e.timestamp ← obtainServiceTimeStamp(lstState.lastInsertTimestamp)
5:     e.clientSession ← getClientSessionID()
6:     lstState.lastInsertTimestamp ← e.timestamp
7:     SERVICE.insert(Lst, ElementId, e)
8:     lstState.insertSet ← addElementToInsertSet(e, lstState.insertSet)
9:     listStates[Lst] ← lstState

10: function get(Lst) do
Read
11:    lstState ← listStates[Lst]
12:    sl ← SERVICE.get(Lst)
Transform
13:    sl ← orderByTimestamp(sl)
14:    sl ← addMissingSessionElementsToSL(sl, lstState.insertSet, lstState.lastSessionTimestamp)
15:    sl ← purgeOldSessionElementsFromSL(sl, lstState.lastSessionTimestamp)
16:    sl ← subList(sl, 0, N)
Store
17:    lstState.lastSessionTimestamp ← getLastSessionTimestamp(sl)
18:    lstState.insertSet ← purgeOldElements(lstState.insertSet, lstState.lastSessionTimestamp)
19:    listStates[Lst] ← lstState
20:    return removeMetadata(sl)
```

This definition, however, does not consider the case where only the N elements of a list are returned by a get operation. In this case, some writes of a given client may not be present in the result if more than N other insert operations have been performed (by client c or any other client). Considering that the list must hold the session elements in the order they were issued, a RYW anomaly happens when a get operation returns an older write performed by the client but misses a more recent one. More formally, given two writes x, y over list L executed in the same client session, where x was executed before y, an anomaly of RYW happens in a get that returns S when $\exists\, x, y \in W : x \prec y \wedge y \notin S \wedge x \in S$.

Algorithm 2 presents our algorithm for providing RYW. To avoid the anomaly described above, the idea is to store, locally at the client, all elements that are inserted by the local client in the list and add them to the result of get operations. In the insert operation, the inserted element is stored locally by the client (line 8). Additionally, our algorithm stores some metadata in the object before performing the insert operation over the service (lines 3 – 4). This information represents, respectively, the identifier of the element and a timestamp

for the insert operation (from the perspective of the service, and retrieved as described in Section 4.1). The element identifier is used to uniquely identify the writes. The timestamp and the element identifier allow for totally ordering all entries in the **insertSet**, with the order being approximately that of the real-time order of execution. The operation in line 4 also checks if the timestamps retrieved from the service in the same session are monotonically increasing, and, if not, enforces that property by overwriting the returned timestamp with an increment of the most recent one, that is stored in the **lastInsertedTimestamp**; this is important to avoid reordering events from the same session in case the timestamp provided by the server does not increase monotonically for some reason. Note that it is not necessary to always obtain the time from the service before each insert operation, we can retrieve the time only on the first time the service is accessed and calculate the clock delta by taking into consideration the passing of time from the perspective of the clock in the client machine, and then use this value and the client clock to calculate a new timestamp.

All of our operations are split into three blocks: read, transformation, and store. In particular, for executing a get operation (line 10) our algorithm starts by executing the get operation over the service (line 12), in the read block of the algorithm. Then, the returned list (*sl*) is ordered (line 13) and all elements of the local **insertSet** that are missing in the list are added to the list, keeping the list ordered (line 14), this is done in the transformation block of the algorithm. Before returning the most recent N elements (with no metadata) (line 20), our algorithm removes old session elements from the *sl* list and updates the **lastSessionTimestamp** variable with the timestamp of the oldest element of the client session returned to the client (lines 17), this last operation is done in the store block of the algorithm.

To avoid the **insertSet** to grow indefinitely, we use the timestamp of each element to remove from the **insertSet** any element older than **lastSessionTimestamp** (line 18). We also need to include the session identifier in the metadata of each element to avoid old elements of the session to reappear (line 15).

Algorithm 3: Monotonic reads

```
1:  function insert(Lst, ElementID, Value) do
2:     Element e ← (ElementID, Value)
3:     e.timestamp ← obtainServiceTimeStamp()
4:     SERVICE.insert(Lst, ElementID, e)

5:  function get(Lst) do
Read
6:     lstState ← listStates[Lst]
7:     sl ← SERVICE.get(Lst)
Transform
8:     sl ← orderByTimestamp(sl)
9:     sl ← addLocalViewElementsToSL(sl, lstState.localView)
10:    sl ← purgeOldElementsFromSL(sl, lstState.lastTimestamp)
11:    sl ← subList(sl, 0, N)
Store
12:    lstState.lastTimestamp ← getLastTimestamp(sl)
13:    lstState.localView ← addNewElements(sl, lstState.localView)
14:    lstState.localView ← purgeOldElements(lstState.localView, lstState.lastTimestamp)
15:    listStates[Lst] ← lstState
16:    return removeMetadata(sl)
```

4.3.2 Monotonic Reads

This session guarantee requires that all writes reflected in a read are also reflected in all subsequent reads performed by the same client. To define this in our scenario where a truncated list of N recent elements is returned, we say that Monotonic Reads (MR) is violated when a client c issues two read operations that return sequences S_1 and S_2 (in that order) and the following property holds: $\exists x, y \in S_1 : S_1(x) \prec S_1(y) \wedge y \notin S_2 \wedge x \in S_2$, where $S_1(x) \prec S_1(y)$ means that element x appears in S_1 before y.

To avoid this anomaly, our algorithm (presented in Algorithm 3) resorts to the **localView** variable to maintain information regarding the elements (and their respective order) observed by the client in previous get operations. Therefore, when the client issues a get operation, our Middleware issues the get command over the centralized service and then updates the contents of *sl* with the elements that are in the **localView** that are missing. The algorithm terminates by returning to the client the first N elements in *sl*. These elements are exposed to the client without any of the metadata added by our algorithms.

Similar to the previous discussed algorithm, the size of the **localView** can grow indefinitely. To avoid this, the insert operation associates to each element a timestamp obtained from the service and removes from the **localView** all

Algorithm 4: Monotonic Writes

```
1:  function insert(Lst, ElementID, Value) do
2:      lstState ← listStates[Lst]
3:      Element e ← (ElementID, Value)
4:      e.clientSession ← getClientSessionID()
5:      e.sessionCounter ← lstState.insertCounter++
6:      SERVICE.insert(Lst, ElementID, e)
7:      listStates[Lst] ← lstState

8:  function get(Lst) do
Read
9:      lstState ← listStates[Lst]
10:     sl ← SERVICE.get(Lst)
Transform
11:     sl ← sortSessionSequences(sl)
12:     sl ← removeElementsWithMissingDependencies(sl)
13:     return removeMetadata(sl)
```

elements with a timestamp smaller than the timestamp of the last element returned to the client (**lastTimestamp**). When the client issues a get operation, we start by executing the get operation over the service (line 7), in the read block of the algorithm. Then, the returned list (*sl*) is ordered (line 8) and all elements in the **localView** that are missing in the list are added to the list, keeping it ordered (line 9). This is done in the transformation block of the algorithm. Before retuning to the client, we remove old elements, elements that are below **lastTimestamp** (line 10), resize *sl* and update the list local state. This update is done in the store block of the algorithm.

4.3.3 Monotonic Writes

This session guarantee requires that writes issued by a given client are observed in the order in which they were issued by all clients. More precisely, if W is a sequence of write operations issued by client c up to a given instant, and S is a sequence of write operations returned in a read operation by any client, a Monotonic Writes (MW) anomaly happens when the following property holds, where $W(x) \prec W(y)$ denotes x precedes y in sequence W: $\exists x, y \in W : W(x) \prec W(y) \wedge y \in S \wedge (x \notin S \vee S(y) \prec S(x))$.

However, this definition needs to be adapted for the case where only N elements of a list are returned by a get operation. In this case, some session sequences may be incomplete, because older elements of the sequence may be

left out of the truncated list of N returned elements. Thus, we consider that older elements are eligible to be dropped from the output, provided that we ensure that there are no gaps in the session subsequences and that the write order is respected, before returning to the client. Formally, we can redefine MW anomalies as follows, given a sequence of writes W in the same session, and a sequence S returned by a read: $(\exists x, y, z \in W : W(x) \prec W(y) \prec W(z) \wedge x \in S \wedge y \notin S \wedge z \in S) \vee (\exists x, y \in W : W(x) \prec W(y) \wedge S(y) \prec S(x))$.

Algorithm 4 presents the steps employed by our Middleware to enforce the MW session guarantee. We avoid the anomaly described above by adding metadata to each insert operation (lines $1-7$) in the form of a unique client session identifier (*clientSession* – line 4) and a counter (local to each client and session) that grows monotonically (*sessionCounter* – line 5). This information allows us to establish a total order of inserts for each client session.

This metadata is then leveraged during the execution of a get operation (lines $8-13$) in the following way. After reading the current list from the service (line 10), in the read block of the algorithm, we simply order the elements in the read list (*sl*) to ensure that all elements respect the partial orders for each client session (line 11). Finally, an additional step is required to ensure that no element is missing in any of these partial orders. To ensure this, whenever a gap is found within the elements of a given client session, we remove all elements whose *sessionCounter* is above the one of any of the missing elements, these operations are all done in the transformation block of the algorithm.

The get operation returns the contents that are left in the list *sl* without the metadata added by our algorithms (line 13). Note that in this case we might return to the client a list of elements with a size below N. We could try to mitigate this behavior by resorting to the contents of the **localView** as we did in the algorithm that we designed to enforce MR. However, we decided to provide the minimal behavior to enforce each of the session guarantees in isolation.

Algorithm 5: Writes Follow Read

```
1:  function insert(Lst, ElementID, Value) do
2:     lstState ← listStates[Lst]
3:     Element e ← (ElementID, Value)
4:     e.cutTimestamp ← obtainCutTimestamp(lstState.localView)
5:     e.dependencies ← projectElementIdentifiersAndTimestamps(lstState.localView)
6:     e.timestamp ← obtainServiceTimeStamp(lstState.lastInsertTimestamp, lstState.localView)
7:     lstState.lastInsertTimestamp ← e.timestamp
8:     SERVICE.insert(Lst, ElementID, e)

9:  function get(Lst) do
Read
10:    lstState ← listStates[Lst]
11:    sl ← SERVICE.get(Lst)
Transform
12:    sl ← orderByTimestamp(sl)
13:    sl ← removeElementsWithMissingDependencies(sl)
14:    cutTimestamp ← highestCutTimestamp(sl)
15:    sl ← removeElementsBelowCutTimestamp(sl, cutTimestamp)
Store
16:    lstState.localView ← addNewElements(ls, lstState.localView)
17:    lstState.localView ← purgeOldElements(lstState.localView)
18:    listStates[Lst] ← lstState
19:    return removeMetadata(sl)
```

4.3.4 Writes Follow Reads

This session guarantee requires that the effects of a write observed in a read by a given client always precede the writes that the same client subsequently performs. (Note that although this anomaly has been used to exemplify causality violations [5, 49], any of the previous anomalies represent a different form of a causality violation [64].) To formalize this definition, and considering that the service only returns at most N elements in a list, if S_1 is a sequence returned by a read invoked by client c, w a write performed by c after observing S_1, and S_2 is a sequence returned by a read issued by any client in the system; a violation of the Writes Follow Read (WFR) anomaly happens when: $\exists w \in S_2 \wedge \exists x, y \in S_1 : S_1(x) \prec S_1(y) \wedge y \notin S_2 \wedge x \in S_2$.

Our algorithm to enforce this session guarantee is depicted in Algorithm. 5. The key idea to avoid this anomaly is to associate with each insert the direct list of dependencies of that insert, i.e., all elements previously observed by the client performing the insert (line 5). Evidently, this solution is not practical, since this list could easily grow to include all previous inserts performed during the lifetime of the system. To overcome this limitation, we associate with each

insert a timestamp based on the clock of the service, that increases monotonically in the session, but with the restriction of being strictly greater than the timestamp of any of its direct dependencies (line 6). Furthermore, we also associate with each insert operation a cut timestamp, that defines the timestamp of its last explicit dependency, i.e., the dependencies registered in the dependency list (line 4). The cut timestamp implicitly defines every element with a lower timestamp to be a dependency of that insert operation. By combining these different techniques, we ensure that the explicit dependency list associated with an insert has at most a value around N elements (which is the size of the **localView** maintained by our Middleware). Note that the dependency list associated to each element, does not need to contain all the information of an element, it can be a list of pairs <$ElementID$, $ElementTimestamp$>, which is the necessary information to identify an insert operation, which is enough to enforce this session guarantee.

Since only N elements of a list are returned by a get operation, the older dependencies may be left out of the sequence that is returned. When this happens, it is safe to consider that these dependencies were dropped from the window that is returned, provided that we ensure that, for each element that is returned, all dependencies that are more recent than the oldest element are also returned.

In the get operation we leverage this metadata to do the following: we start by reading the contents of the list from the service (line 11), in the read block of the algorithm, and then over this list we remove any insert whose dependencies are missing. Thus, we only remove inserts whose missing dependencies have a timestamp above the insert cut timestamp. We then compute a cut timestamp for the obtained list *sl* (line 14) that is the highest cut timestamp among all elements in *sl*. We use this timestamp to remove from *sl* any element whose creation timestamp falls below the computed cut timestamp. These operations are all done in the transformation block of the algorithm. Finally, before returning to the client the elements that remain in *sl* without the additional metadata (line 19) we update and garbage collect old entries from the **localView** (lines 16 – 18), this is done in the store block of the algorithm.

Algorithm 6: Combinations insert operation

```
1:  function insert(Lst, ElementID, Value) do
2:     lstState ← listStates[Lst]
3:     Element e ← (ElementID, Value)
4:     if MW and WFR then
5:        e.lastInsertTimestamp ← lstState.lastInsertTimestamp
6:     if RYW or MR or WFR then
7:        e.timestamp ← obtainServiceTimeStamp(lstState.lastInsertTimestamp, lstState.localView)
8:        lstState.lastInsertedTimestamp ← e.timestamp
9:     if RYW or MW then
10:       e.clientSession ← getClientSessionID()
11:    if MW then
12:          e.sessionCounter ← lstState.insertCounter++
13:    if WFR then
14:       e.cutTimestamp ← obtainCutTimestamp(lstState.localView)
15:       e.dependencies ← projectElementIdentifiersAndTimestamps(lstState.localView)
16:    SERVICE.insert(Lst, ElementID, e)
17:    if RYW then
18:       lstState.insertSet ← addElementToInsertSet(e, lstState.insertSet)
19:    listStates[Lst] ← lstState
```

Similarly to the previous algorithm, we might return a number of elements that is lower than N. In this case, to ensure that we always return N elements, we need to obtain the missing dependencies using a get operation that returns a single element (if supported by the service). In our implementation, we avoided this solution because it is prone to triggering a violation of the API rate limits.

4.3.5 Combining Multiple Session Guarantees

Considering the algorithms to enforce each of the session guarantees discussed above, we can now summarize how to combine them in Algorithm 6 and in Algorithm 7. The insert operation adds the metadata used by each of the individual algorithms according to the guarantees configured by the application developer. Correspondingly, upon the execution of a get operation, our Middleware must perform the transformations over the list obtained from the service (*sl*) prescribed by each of the individual algorithms.

More precisely, the insert operation starts by storing the necessary metadata in the element object before performing the insert operation over the centralized service (lines 2 – 15). Note that, to obtain a correct timestamp,

Algorithm 7: Combinations get operation

```
1:  function get(Lst) do
Read
2:     lstState ← listStates[Lst]
3:     sl ← SERVICE.get(Lst)
Transform
4:     if RYW or MR or WFR then
5:        sl ← orderByTimestamp(sl)
4:     if RYW then
6:        sl ← addMissingSessionElementsToSL(sl, lstState.insertSet, lstState.lastSessionTimestamp)
7:        sl ← purgeOldSessionElementsFromSL(sl, lstState.lastSessionTimestamp)
8:     if MR or WFR and RYW then
9:        sl ← addLocalViewElementsToSL(sl, lstState.localView)
10:       sl ← purgeOldElementsFromSL(sl, lstState.lastTimestamp)
11:    if MW then
12:       if MR then
13:          sl ← removeOldElementsFromSessionSequence(sl,lstState.localView)
14:       sl ← sortSessionSequences(sl)
15:       sl ← removeElementsWithMissingDependenciesMW(sl)
16:    if WFR then
17:       sl ← removeElementsWithMissingDependenciesWFR(sl)
18:       cutTimestamp ← highestCutTimestamp(sl)
19:       sl ← removeElementsBelowCutTimestamp(sl, cutTimestamp)
20:       if MW then
21:          sl ← removeElementsBelowMisssingSessionElement(sl)
22:    sl ← subList(sl, 0, N)
Store
23:    if RYW then
24:       lstState.lastSessionTimestamp ← getLastSessionTimestamp(sl)
25:       lstState.insertSet ← purgeOldElements(lstState.insertSet, lstState.lastSessionTimestamp)
26:    if MR or WFR then
27:       lstState.lastTimestamp ← getLastTimestamp(sl)
28:       lstState.localView ← addNewElements(sl, lstState.localView)
29:       lstState.localView ← purgeOldElements(lstState.localView, lstState.lastTimestamp)
30:    listStates[Lst] ← lstState
31:    return removeMetadata(sl)
```

the operation in line 7 of the algorithm always generates a timestamp that increases monotonically with the number of insert operations and when WFR is selected is also greater than the timestamp of any of its direct dependencies, as described in Section 4.3.4.

The get operation starts by executing the read block, which is the same in all algorithms (lines 2 – 3), then applies the transformation blocks of each algorithm, and before returning to the client executes the store blocks of each algorithm. To simplify the algorithm the store block of Writes Follow Read was changed and now uses the **lastTimestamp** to remove the old elements from the **localView**. In other words, the store block used in this case is similar to the store block of the Monotonic Reads algorithm presented earlier. There were

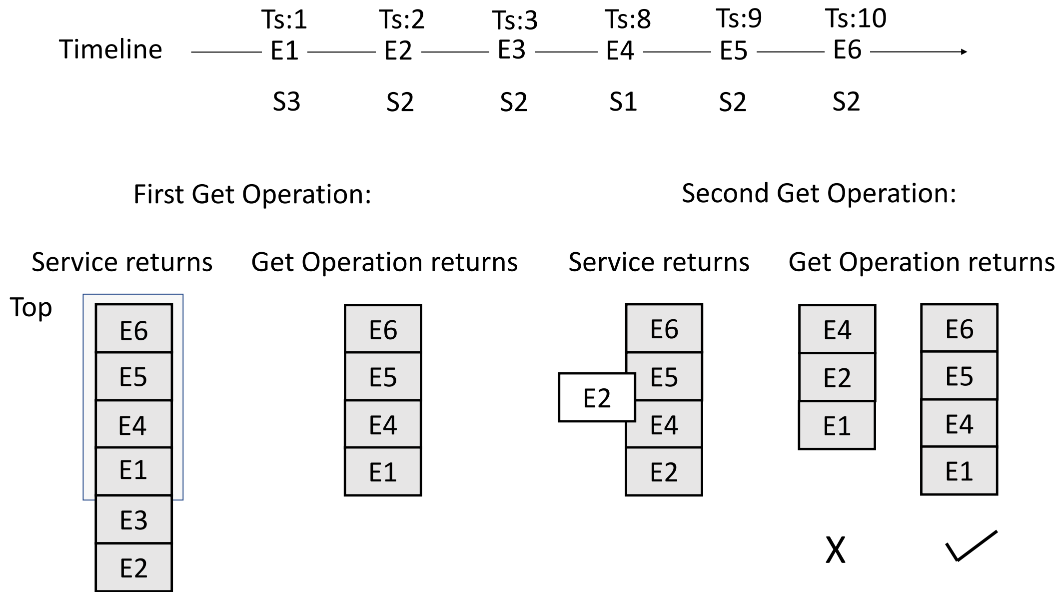


Figure 4.4: Combinations anomaly

other changes in the algorithm that we explain in the following section.

4.3.6 Corner Cases

In this section we describe three corner cases that may arise when the algorithms are working in combination.

To guarantee that the algorithm ensures safety, it is necessary to ensure that the transformation blocks of the algorithms that remove elements from *sl* do not remove elements that might affect the guarantees provided by the transformation blocks of the other algorithms that were executed before. There are two situations where this can occur:

The first situation happens when Monotonic Reads and Monotonic Writes guarantees are selected in combination, In particular, this situation occurs when the **localView** contains elements previously returned to the client, and in the next get operation the service returns elements from a session that were assumed to be truncated previously. This situation is exemplified in Figure 4.4.

In this example, we have a timeline with six elements inserted in a list by session 1, session 2, and session 3. The first get operation returns four elements, one from session 1, the last two inserted in session 2, and another from session 3. Note that the service originally returned all elements and truncated elements E3 and E2, and returned the last two elements inserted in session 2, E4 and E1, which is valid considering the Monotonic Writes property. The next get operation, initiated by session 2, obtains from the service the same three elements at the top, but the last is E2, which was the first element inserted in session 2. In this case, the list returned by the service has an anomaly of the Monotonic Writes property because element E3 is missing. This may happen if the get operation is executed in a different replica of the service, that did not receive the other inserted elements yet. Our algorithm, in this situation is going to merge the **localView**, which contains the elements previously return to the client, with the list returned by the service, and the Monotonic Writes transformation block is going to remove E5 and E6 to eliminate a gap in the sequence of elements added by session 2. This situation causes an anomaly because the get operation is going to return elements that preceded E5 and E6 previously, but misses E5 and E6. To eliminate this anomaly, it is necessary to guarantee that the new elements inserted in *sl* do not create a gap below the oldest element from a session subsequence that had been previously returned to the client. To avoid this situation, before executing the Monotonic Writes transformation block, we use the **localView** to identify the previous session's subsequences returned to the client, we then remove from *sl* the new elements of each previous subsequence that are older than the last element of each subsequence previously returned to the client, this is done in line 13 of Algorithm 7. In the example the get operation is going to return the same elements that were previously returned to the client, which is safe. This happens because E2 was the only new element returned in the last request from the service, and was removed from *sl*, which means that the elements that are going to be returned are the elements that were already stored in the **localView**.

The second situation happens when Read Your Writes and Writes Follow

Reads are selected together. To simplify the explanation, lets assume the following scenario: a client in a session issues a get operation that returns the elements E1, E2, and E3 from other sessions that are dependencies of each other, E1<E2<E3. Then the client inserts E4, a new element in the list (in this case, E3, E2, and E1 are dependencies of E4). The client then issues another get operation and the service retrieves only E3 and E1, and the Read your Writes transformation block is going to insert E4 in *sl*, because E4 is missing. In this case, Writes Follow Reads transformation block removes E4 from *sl* because the E2 dependency is missing, this situation causes an anomaly, because the get operation should return E4 and all of its dependencies. To guarantee this, we need to ensure that the dependencies of E4 are in *sl*, to this end, the elements in **localView** i.e., the elements previously returned to the client, must be included in *sl*. To achieve this, the Monotonic Reads transformation block is executed before Writes Follow Reads transformation block, which will guarantee that all missing elements are added to *sl*, avoiding the exclusion of E4 by the transformation block of the algorithm that enforces Writes Follow Reads. Doing this guarantees that E4 and the respective dependencies are returned to the client.

Finally, the last corner case happens when Monotonic Writes and Writes Follow Reads are enforced together. To simplify the explanation let's assume the following scenario: a client in a session issues a get operation and then inserts two elements in sequence. In this case, the dependencies of the two elements are the elements retrieved in the get operation. If a client in another session receives a list that contains several elements, including the dependencies and only the last element of the sequence, an anomaly occurs, because the first element of the sequence is missing and its successor and the dependencies of the two are present. To avoid this situation we associate to each element the timestamp of the previous element inserted in the session, and at end of the Writes Follow Reads transformation block we remove all elements with a timestamp below the missing element. In this case we return a suffix of the list generated by the execution of both transformation blocks of the algorithm.

This behavior emerges as a consequence of being able to switch on and off

individual session guarantees, we could have interpreted this is as an example of the subtle semantic distinctions of the various session guarantees however, we decided to interpret this behavior as something that should be avoided, and adapted our algorithms to preclude this situation.

According to our analysis of the algorithms these are the only possibilities for this type of situations where one algorithm will break the conditions that are necessary for the correctness of another algorithm. In Section 4.6 we present the correctness arguments detailing why this is the case.

4.3.7 Progress

Our goal is to provide safety to applications that are using Internet services; however we also need to guarantee progress, i.e., we need to ensure that the algorithms alone or in combination, in a session, will return new elements. When we combine the algorithm of Monotonic Reads with Monotonic Writes or Writes Follow Reads, it becomes more challenging to ensure progress when a client issues a sequence of get operations over the same list (i.e., data objects). The problem arises when a get operation receives from the service a list with a sequence of elements that are more recent than the previous elements returned to the client. This situation can produce a gap between the two lists and Monotonic Writes or Writes Follow Reads algorithms may remove all new elements to ensure safety, causing subsequent get operations to return the same elements in every request. This situation is illustrated in Figure 4.5. In this example we have a timeline with seven elements inserted in a list by session 1 and session 2. The first get operation returns the first two elements inserted in the list, one from session 1 and another from session 2; Then the next get operation, initiated by the same client, obtains from the service the last two elements inserted, E6 and E7, in the context of session 2. In this situation, the Monotonic Writes algorithm detects that E4 from session 2 is missing and returns E1 and E2.

In order to avoid this problem, we execute the transformation block of the get operation and test if the produced list contains the same elements that were previously returned to the client. If the list contains the same elements and there is a time gap between this list and the list returned by the service, we run

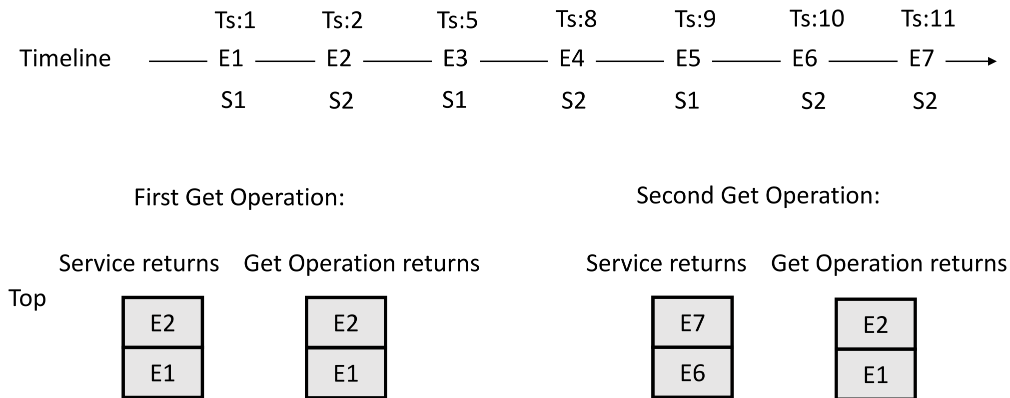


Figure 4.5: Time Gap

again the transformation block for the service list, but before, we remove all elements from the **localView**, and we use the timestamp of the oldest element in service list as **lastTimestamp**. After executing the get operation we compare the number of elements of the two produced lists and return the list with more elements or the list with the most recent elements if they have the same number of elements. To guarantee progress, we also need to avoid situations where the previous list returned to the client is always bigger than the list with the most recent elements. To do this, we define the maximum number of times that the old list can be returned. This value serves as a configuration parameter for our middleware layer. Leveraging this solution guarantees that in the example described previously we are going to return E7 and E6. Note that this is the same behavior provided by a service that ensures the properties mentioned above in combination, the first get operation returns E2 and E1 and the second get operation is going to return E7 and E6 because the service contains the seven elements and the get operation only returns the two elements at the top.

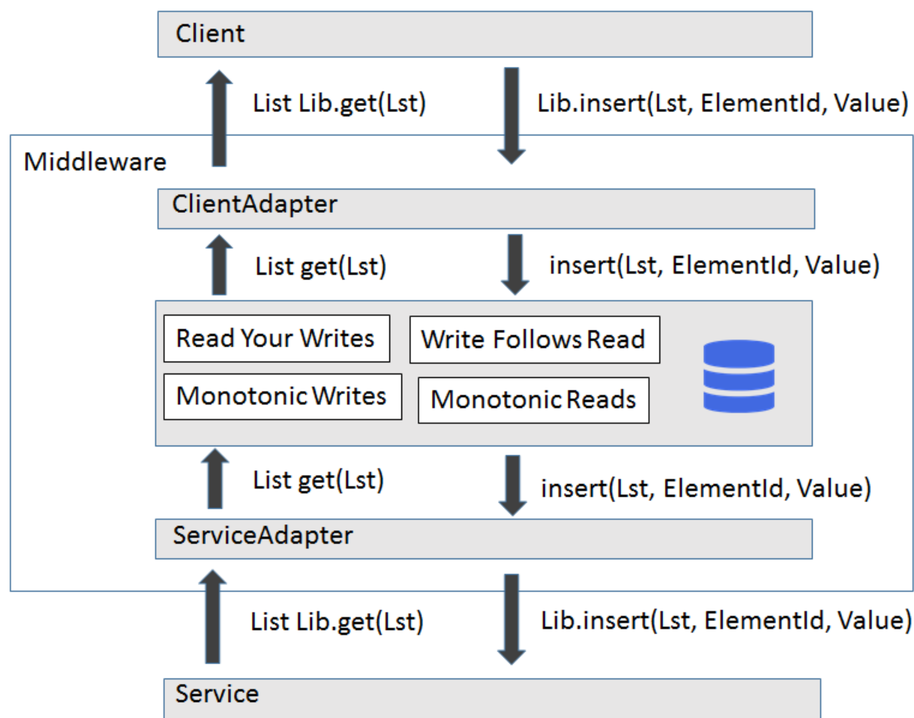


Figure 4.6: Middleware with adapters

4.4 Middleware Design

Another goal we have is to allow our solution to be generic and easy to adapt, to allow accessing any Internet service with a public API, and give the developers the same library interface to work with. To this end, we introduced two adapter layers that must be provided (see Figure 4.6) to execute our middleware in this context. These layers capture the API calls performed by the client application and translate them to a standard API exposed by our Middleware (ClientAdapter), and translate the calls to the centralized service performed by our Middleware into API calls to the library used to interact with the service (ServiceAdapter), respectively. The adapters themselves are quite straightforward to write, and we believe most developers will be able to easily write new adapters to use our Middleware in combination with different libraries for accessing other online services.

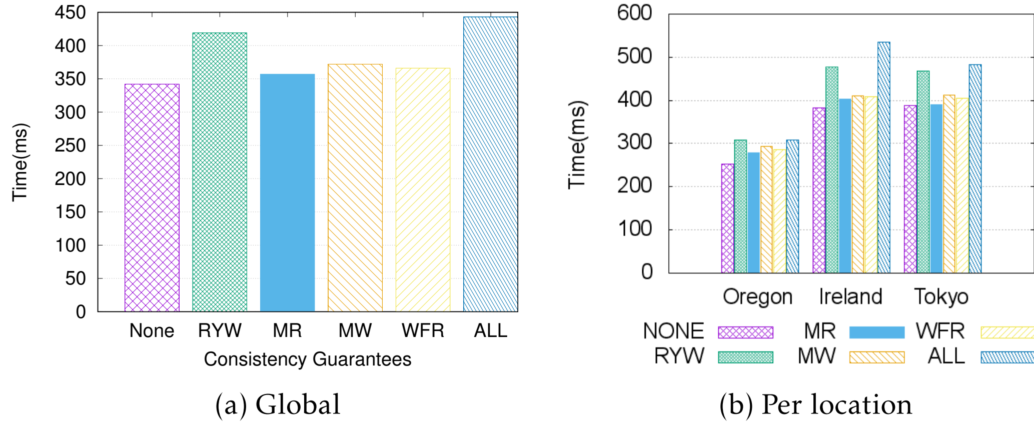
4.5 Evaluation

In this section we present the experimental evaluation of our Middleware, which compares the client-perceived performance obtained when using our Middleware to provide each of the session guarantees in isolation and their combination (i.e, enforcing all four session guarantees). In our experiments we used a prototype of our Middleware and the evaluation was made using two different geo-replicated online services. First, to illustrate the benefit of our Middleware when designing third-party applications that interact with online social networks we have used Facebook's public API. Then, to illustrate the operation of our Middleware when interacting with a service that imposes fewer restrictions on the number and timing of client operations, we experimented with a geo-replicated deployment of the Redis data store managed by ourselves.

Our evaluation focuses on assessing the overhead that results from the use of our middleware, in terms of client perceived latency (for insert and get operations), the communication overhead due to the inclusion of additional metadata, and the storage overhead, namely due to the need for our Middleware to locally maintain some information about previous operations performed by the client. Our prototype of the Middleware layer was implemented in the Java language. To interact with the two services that we explore in this work, we resorted to the *restFB* library for Facebook [59], and the *Jedis* library for interacting with Redis [43], for each library we implemented the respective adapters explained previously in Section 4.4.

4.5.1 Facebook Results

We have conducted our experiments with Facebook by using YCSB [4] to emulate clients using Facebook to post messages to a group feed and reading the contents of that group feed. To emulate such clients spread across the World, we run three independent YCSB instances in three different locations using Amazon EC2 [6] instances in Oregon, Ireland, and Tokyo. Each YCSB instance uses 10 threads, emulating a total of 10 independent clients, for a total of 30



(a) Global

(b) Per location

Figure 4.7: Latency of Get Operation in Facebook

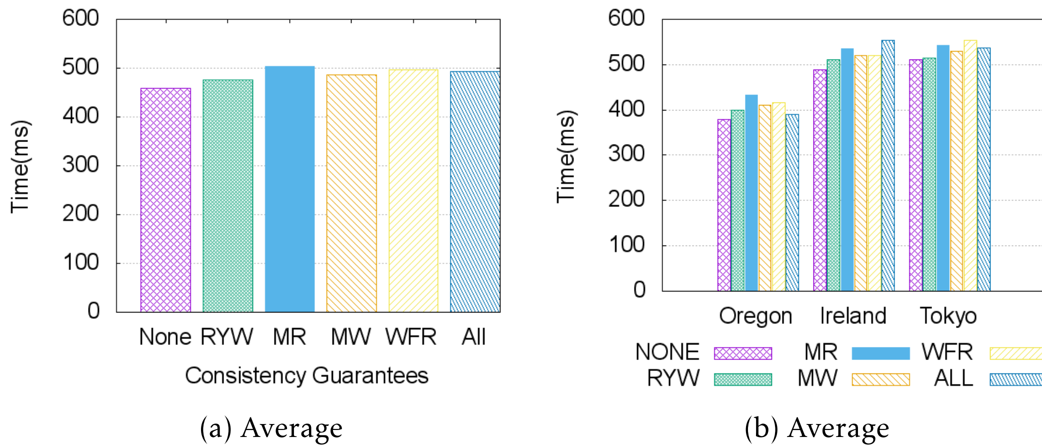
clients across the World. Each emulated client has an independent instance of our Middleware. To accommodate the rate limits of Facebook's public API, we impose a maximum of 15 requests per second per YCSB instance.

Each experiment reported in this section was executed 7 times, and different consistency guarantees were rotated along experiments, such that each different consistency guarantee had experiments running on different time periods of the day. This was done to remove experimental noise due to contention on the Facebook servers, (e.g., to compensate for the activity of real users of the system). The workload executed by clients was a mix of 50% insert and 50% get operations. The Middleware was configured to have $N = 25$, meaning that each get retrieves at most 25 elements from the feed. Experiments reported in this section report the aggregated observations of 53,119 insert and get operations.

4.5.1.1 Latency

We start by observing the latency of operations in Facebook when accessing the service directly through the library (labeled in the plots as None) and when using our Middleware to enforce each of the session guarantees in isolation and all of the session guarantees (labeled in the plots as All).

Figure 4.7 reports the latency observed for get operations, for all clients and per location of the client. Figure 4.7a shows that our Middleware introduces a small increase in the latency of get operations with a maximum increase of approximately one hundred milliseconds. Not surprisingly the overhead is at



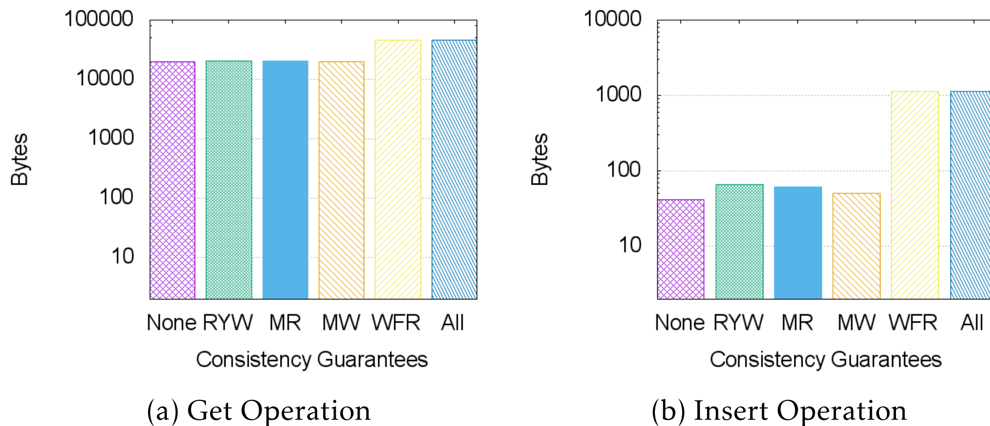
(a) Average (b) Average

Figure 4.8: Latency of Insert Operation in Facebook

its maximum when all session guarantees are being enforced by our Middleware which is explained by a combination of the additional metadata carried in each element, and the processing cost of the Middleware to perform the enforcement of each individual session guarantee.

When observing the distribution of latency for requests according to the region where the client is located (Figure 4.7b), we note the same relative distribution in the results, with overall lower latency values for the clients in Oregon. This is explained by the latency of those clients towards the Facebook servers, which is notoriously smaller as confirmed by measuring the latency when using the client library directly. Another noteworthy aspect of Figure 4.7b is that the observed latency has a visible variation, both across and even within different client locations. This suggests that the latency overhead in these cases may suffer from a noticeable variability due to external factors which are related with the architecture and deployment of such a large-scale real world application.

Figure 4.8 reports average latency results for the insert operation for all clients and per client location. The results reported in Figure 4.8a show that globally the latency penalty incurred by the use of our Middleware is again modest, with a maximum increase of at most 50 milliseconds. The individual session guarantee with the largest increase in latency is Monotonic Reads.



(a) Get Operation (b) Insert Operation

Figure 4.9: Communication overhead in Facebook

Considering the latency values observed in different locations reported in Figure 4.8b, we can observe the same pattern previously observed, where the latency experienced by clients in Oregon is lower compared with the remaining locations. This is expected, since this can be explained by the latency experienced by the client to contact the Facebook service in that concrete location when compared with the remaining locations used in our experimental work.

4.5.1.2 Communication Overhead

We now study the communication overhead imposed by our Middleware by observing the average size of messages exchanged between clients and the service. Figure 4.9 reports these results for each of the session guarantees and for their combination, compared with the use of the library without our Middleware, for both get and insert operations. The results in Figure 4.9a show that the overhead introduced by our Middleware is noticeable for get operations when Writes Follow Reads and the combination of all session guarantees are enforced. This happens because most of the payload in these messages are the multiple elements of the list that are returned, and in these cases each element contains the explicit dependencies. Note that each element also contains the metadata that Facebook associates to each post.

The same pattern occurs for the insert operations, as reported in Figure 4.9b. In this case, each message contains only a single element to be added, the increase in message size is quite noticeable when the Middleware is enforcing

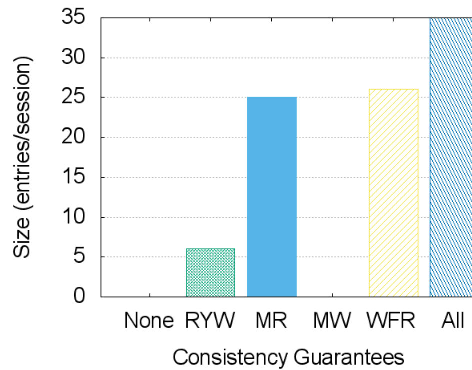


Figure 4.10: Local storage overhead for Facebook

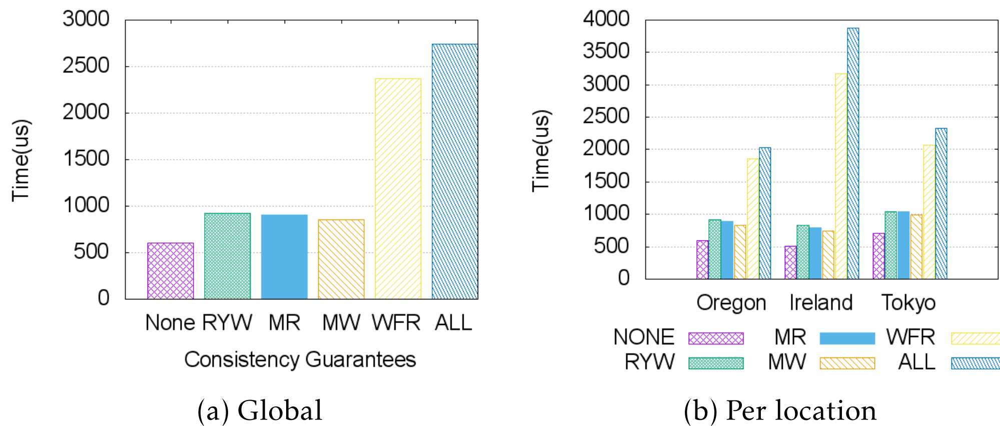
Writes Follow Reads and the combination of all session guarantees. This happens due to the cost of sending the explicit dependencies of each inserted element, which can account to 25 unique element identifiers and their timestamps. The remaining session guarantees, in contrast, have a modest overhead of only a few tens of bytes.

4.5.1.3 Local Storage Size

Finally, Figure 4.10 reports the storage cost in terms of elements stored locally by our Middleware for enforcing each of the session guarantees and their combination. For completeness, we also provide the results for the None configuration, which, as expected, is zero. This is used as a sanity check for our results. Monotonic Writes do not require any form of local storage, and therefore have no local storage overhead. In contrast, the remaining session guarantees do exhibit some low storage overhead due to their need to maintain elements stored in the insertSet and localView data structures. As expected, when providing all of the session guarantees the local storage has more entries, leading to additional overhead. This happens because the number of entries is the sum of the elements in the insertSet and in the localView.

4.5.2 Redis Results

We also conducted experiments using the Redis data storage system. To this end, we deployed Redis with its replication enabled across machines scattered



(a) Global

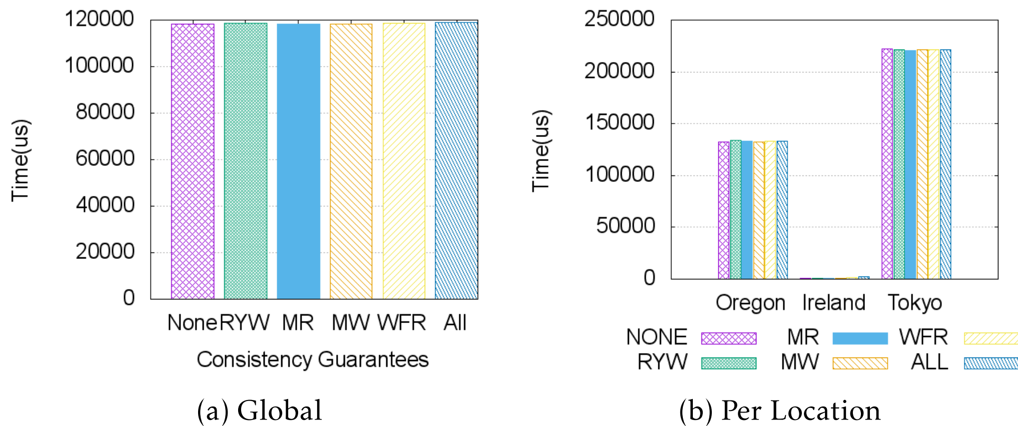
(b) Per location

Figure 4.11: Latency of Get Operation in Redis

in three Amazon EC2 regions: Oregon, Tokyo, and Ireland. Redis uses a master-slave replication model, and we have deployed the master in Ireland and two slaves in each region, for a total of 7 replicas. We used m1.large instances to run the master and YCSB and m1.medium instances to run the slaves. YCSB was executed in the same three regions of Amazon EC2 used in the previously reported experiments, with each YCSB instance running 10 threads that execute operation in a closed loop. Each thread has its own instance of the Middleware. All operations access the same list object stored in Redis, with the read operation being executed in one of the slave replicas of the region, selected randomly. For each algorithm, we run our experiments 6 times for 60 seconds with an interval of four minutes between runs. Similar to the experiments conducted with Facebook, YCSB was configured to execute a workload composed of 50% insert and 50% of read operations. Again, we set N to be equal to 25. The experiments reported in this section aggregate the results from executing a total of $21,285,291$ insert and get operations.

4.5.2.1 Latency

Figure 4.11a presents the average latency of get operations. The results show that our middleware introduces a very small overhead, on the order of microseconds, for Read Your Writes, Monotonic Reads, and Monotonic Writes. In Writes Follow Read and when all session guarantees are enforced, there is an increase of approximately one to two milliseconds because the algorithms have



(a) Global (b) Per Location

Figure 4.12: Latency of Insert Operation in Redis

to check the dependencies and process all the metadata information associated with the various objects returned and stored locally. The results of Figure 4.11b, which details the values observed in each region, show the same pattern across all regions, however the latency for reading data using a client in Ireland is higher than in other locations. This can be explained by the fact that writes in Ireland are much faster that in other locations, due to the proximity to the master replica, which causes the total number of read operations that are executed to be higher in Ireland than in other locations, thus leading to a higher load, which results in a higher latency for executing operations particularly, get operations.

In contrast to the experiments for the Facebook service, the observed latencies are much more predictable in this deployment. This confirms the expectation that a real-world service leads to qualitatively different results from a controlled experiment.

Figure 4.12a reports average latency of the insert operation across all locations. In this case the latency is almost the same across all cases, but if we look at Figure 4.12b we see that, in Ireland, latency values are much smaller. This is again justified by the location of the master replica in Ireland and the fact that all clients are issuing their write operations to the (same) master replica. Figure 4.13 reports the latencies in Ireland, which again show a similar pattern to the one observed for Get operations.

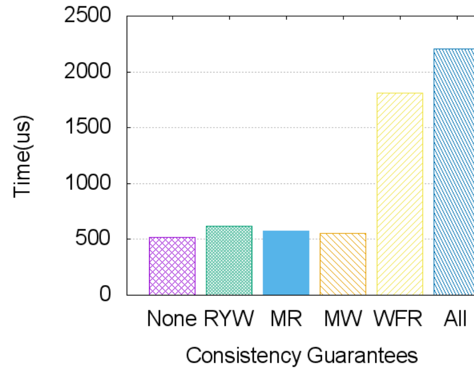
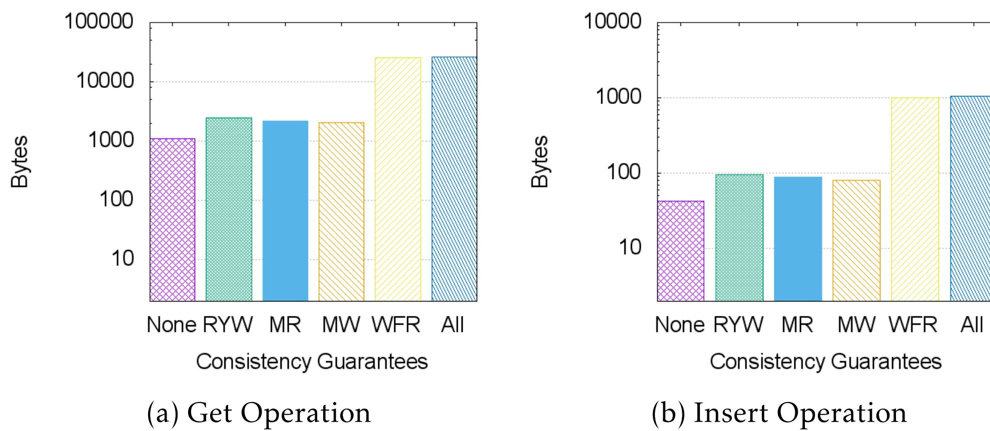


Figure 4.13: Latency of Insert Operation in Redis in Ireland



(a) Get Operation

(b) Insert Operation

Figure 4.14: Communication overhead in Redis

4.5.2.2 Communication Overhead

In terms of communication overhead imposed by our Middleware, the results in Figure 4.14a and Figure 4.14b show that in the get and insert operations the overhead is more noticeable when enforcing Writes Follow Reads and when employing the combination of all algorithms. This happens due to the overhead associated with managing and communicating the information stored in dependency lists, as discuss previously for the results reported for Facebook.

4.5.2.3 Local Storage Size

To conclude our experimental evaluation of Redis, Figure 4.15 shows that in Monotonic Reads and Writes Follow Reads the number of elements in the localView is around 30, which is higher than $N = 25$. This happens because of the high write throughput, which causes several elements to be assigned the

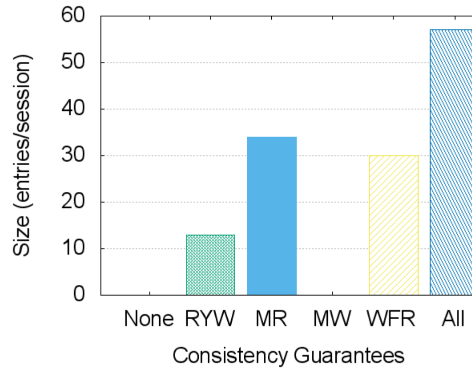


Figure 4.15: Local storage overhead for Redis

same timestamp. In this case, our truncation algorithm allows for the limit to be exceeded in the case of ties. The combinations of all algorithms is also affected by this situation, leading to a higher value around 55. Note that we are showing the average of the highest value registered for each independent client session at any time during its execution.

4.6 Arguments of Correctness

In this section we present the arguments of correctness of our algorithms, in terms of their ability to guarantee session properties individually and in combination.

4.6.1 Read Your Writes

In this section, we are going to argue that the algorithm guarantees Read Your Writes. Recall the definition of an anomaly of Read Your Writes:

The get operation returns an older element inserted by the client before a more recent one, more precisely, there exist two elements x, y inserted over list L in the same client session, in this order(x then y), and a get returns S, the top of the list, and $y \notin S \land x \in S$.

To guarantee that this anomaly does not occur, it suffices to return a set (the truncated list of elements that are returned to the application) such that, when we project the elements from that list that belong to the current client session, we obtain a suffix of the sequence of elements inserted in that session. This

suffices to prevent the anomaly since a property of a suffix is that, when an element is present, all its successors are also present. To ensure this, we assign each element a timestamp that increases monotonically with the order of the operations within the session, and we return all the session elements larger or equal to the timestamp of the oldest session element previously returned to the client, we call this timestamp **lastSessionTimestamp**, the way that this timestamp is set does not affect correctness, but it may affect the length of the window that is returned. In practice, we decided to set it with the timestamp of the oldest session element previously returned to the client, because we assumed that older elements were dropped from the window. Note that the elements returned to the client are from several session, and that sometimes it will be impossible to return all session elements with a timestamp above **lastSessionTimestamp**, because they were truncated, this situation does not affect correctness because we are still returning a suffix of the sequence of elements inserted in the session.

In more detail, in line 12 of Algorithm 2, when the client issues a get operation, the service returns a sublist of elements, that sublist may contain x, a session element above **lasSessiontTimestamp** and miss y an element inserted after x. To avoid this situation, the get operation has to insert the missing elements in the sublist. To this end, the insert operation stores the suffix of the sequence of elements inserted in the session in the **insertSet**, and then the algorithm uses the **insertSet** and the **lasSessiontTimestamp** to detect the missing session elements. Finally, when the algorithm detects that an element is missing, the element is copied from the **insertSet** to the sublist.

In order to prevent returning session elements that do not belong to the suffix, in line 15 of the algorithm, we remove from the sublist all session elements with a timestamp below **lastSessionTimestamp**. To detect these elements in the sublist, in line 5 of the algorithm, we associate to each element a session identifier in the insert operation, we need this session identifier because the service can return session elements that do not belong to the suffix and we only have information about the last session elements inserted in the session, that are in the **insertSet**.

The remainder of the algorithm steps do not break this property, and therefore we can conclude that a violation of the session guarantee does not appear in the newly produced trace.

4.6.2 Monotonic Reads

In this section, we are going to argue that the algorithm guarantees Monotonic Reads. Recall the definition of an anomaly of Monotonic Reads:

When a client c issues two get operations that return sequences S_1 *and* S_2 *(in that order) and the following property holds:* $\exists x, y \in S_1 : S_1(x) \prec S_1(y) \wedge y \notin S_2 \wedge x \in S_2$, *where* $S_1(x) \prec S_1(y)$ *means that element x appears in* S_1 *before y.*

To guarantee that this anomaly does not occur, it suffices to return a set (the truncated list of elements that are returned to the application) such that, we obtain a suffix of the sequence of elements previously returned to the client. Similar to the case of RYW, this suffices to prevent the anomaly since a property of a suffix is that, when an element is present, all its successors are also present. To ensure this, we assign each element a timestamp, and we return all elements larger or equal to the timestamp of the oldest element previously returned to the client, we call this timestamp **lastTimestamp** and it is set just before returning to the client. Again similar to RYW, the way that this timestamp is set does not affect correctness, but it may affect the length of the window that is returned. In practice, we decided to set it with the timestamp of the oldest element previously returned to the client, because we assumed that older elements were dropped from the window.

In more detail, in line 7 of Algorithm 3 when the client issues a get operation, the service returns a sublist of elements, that sublist may contain x, an element previously returned to the client, that is above **lastTimestamp** and miss y, an element returned in a get operation after x. To avoid this situation, the get operation has to insert the missing elements in the sublist. To this end, in line 13 of the algorithm, we store the suffix of the sequence previously returned to the client, in the **localView**. Then, in line 9 of the algorithm, when the algorithm detects that an element is missing, it copies that element from the **localView** to the sublist. In order to prevent returning old elements that do not belong to the suffix, in line 10 of the algorithm, we remove from the sublist all elements with a timestamp below **lastTimestamp**.

The remainder of the algorithm steps do not break this property (enforced by the actions we just described) and therefore, we can conclude that a violation

of the session guarantee cannot appear in the newly produced return value.

4.6.3 Monotonic Writes

In this section, we are going to argue that the algorithm guarantees Monotonic Writes. Recall the definition of an anomaly of Monotonic Writes:

The get operation returns a subsequence of elements from a session in a different order they were issued or with gaps, more precisely, given a sequence of writes W in the same session, and a sequence S returned by a read: $(\exists x, y, z \in W : W(x) \prec W(y) \prec W(z) \land x \in S \land y \notin S \land z \in S) \lor (\exists x, y \in W : W(x) \prec W(y) \land S(y) \prec S(x))$.

To guarantee that this anomaly does not occur, it suffices to return a set (the truncated list of elements that are returned to the application) such that, when we project each session subsequence they are ordered by insertion order and without gaps. To ensure this, we assign each element a unique client session identifier and a counter that increases monotonically in the session, and we return a list with the session elements ordered and without gaps.

In more detail, in line 10 of Algorithm 4, when the client issues a get operation, the service returns a sublist of elements, that sublist may contain x and z, from the session subsequence $x \prec y \prec z$, in this case y is missing from the subsequence returned by the service. To avoid returning a subsequence with a gap, the get operation needs to order the subsequences and remove the existing gaps. To this end, in line 11 of the algorithm, the get operation orders all elements by session counter and, in line 12 of the algorithm, when it detects a gap, removes the elements with a session sequence number above the gap from that subsequence, in this case this entails removing z.

The remainder of the algorithm steps do not break this property that was enforced by the actions we just described (i.e., it will not add the removed elements to the list in the reply to the client) and therefore we can conclude that a violation of the session guarantee does not appear in the newly produced trace.

4.6.4 Writes Follow Reads

In this section, we are going to argue that the algorithm guarantees Writes Follow Reads. Recall the definition of an anomaly of Writes Follow Reads:

*If S_1 is a sequence returned by a get operation invoked by client c, w a write performed by c after observing S_1, and S_2 is a sequence returned by a read issued by any client in the system; a violation of the Writes Follow Reads (*WFR*) anomaly happens when:* $\exists w \in S_2 \land \exists x, y \in S_1 : S_1(x) \prec S_1(y) \land y \notin S_2 \land x \in S_2$.

To guarantee that this anomaly does not occur, it suffices to return a set (the truncated list of elements that are returned to the application) such that, it contains for each element, the list of dependencies, i.e, all elements previously observed in the session where the element was inserted. Note that the list is truncated since it is impossible to keep the entire dependency history, and therefore we only keep the dependencies whose timestamp is larger or equal to the timestamp of the oldest element that is going to be returned.

To achieve this property, while avoiding returning a set that misses an element in a chain of dependencies (e.g., because that element was removed in the context of the algorithm execution), we assign a timestamp to each element that is implemented with the same rules as logical clocks [47], this timestamp increases monotonically with the order of the insert operations within the session and is larger than the timestamp of the most recent element returned to the client. Logical clocks guarantee the partial ordering between elements defined by the happens-before relations [47], this is necessary to guarantee that the elements that were truncated from the list, are not part of the dependencies that should be returned to the client.

We also need to know the elements that form those dependencies. For this purpose, we associate with each element a list with its dependencies. To create this list, we store the elements returned previously, in the **localView**. To avoid having a large list of dependencies, we truncate the dependency list to N elements and we associate to each inserted element the timestamp of the oldest dependency (the element with the lowest timestamp in the **localView**), we call this timestamp **cutTimestamp**, and it implicitly defines that every element with a lower timestamp is a dependency (which while being conservative is

correct).

In more detail, when the service returns a sublist of elements and that sublist contains w and x, but misses y, where w is an element inserted in a session, and y was previously observed in that session and $x \prec y$, then the get operation has to detect that y is missing and remove w from the sublist. To guarantees this, in line 13 of Algorithm 5, we go through the sublist. Starting at the element with the lower timestamp and verify if the dependencies of each element are contained in the list and, if not, the algorithm removes that element.

Note that the previous steps of the algorithm may fail to detect that x and y are dependencies of w, if they were truncated from the dependencies list of w. To guarantee that this situation does not create an anomaly, in lines 14 of the algorithm, we choose the element with the highest **cutTimestamp** in the sublist and remove all elements with a timestamp below **cutTimestamp**. This guarantee, that the sublist returned, always contains for each element, the dependencies with a timestamp larger or equal to the timestamp of the oldest element that is going to be returned, yielding a correct response to the client.

The remainder of the algorithm steps do not break the property enforced by the actions we just described, and therefore we can conclude that a violation of the session guarantee does not appear in the newly produced trace.

4.6.5 Combining Multiple Session Guarantees

In this section, we are going to argue that the algorithms present previously can guarantee the four session properties in combination, when the algorithm executes the transformation blocks of each session property in sequence. The sequence starts with the transformation block from Read Your Writes, followed by the same block of the Monotonic Reads algorithm, then Monotonic Writes and finally, with the Writes Follow Reads transformation block. We are going to show that the execution of each transformation in sequence does not create a consistency anomaly, of the type, of the previously transformations blocks executed.

4.6.5.1 Read your Writes and Monotonic Reads

In this section, we are going to argue that the algorithm guarantees Read Your Writes after applying Monotonic Reads transformation block. Recall the definition of an anomaly of Read Your Writes:

The get operation returns an older element inserted by the client before a more recent one, more precisely, there exist two elements x, y inserted over list L in the same client session, in this order(x then y), and a get returns S, the top of the list, and $y \notin S \wedge x \in S$.

To guarantee that this anomaly does not occur, the strategy used by Read Your Writes is to return a set (the truncated list of elements that are returned to the application) such that, when you project the elements from that list that belong to the current client session, you obtain a suffix of the sequence of elements inserted in that session. To preserve this, the Monotonic Reads transformation block has to return a suffix of the session elements. To this end, the transformation cannot remove session elements that break the suffix. This is guaranteed because the operation in line 10 of Algorithm 7, only removes elements below **lastTimestamp**, i.e, the timestamp of the oldest element previously returned to the client, which guaranties a suffix. It is also guaranteed that the operation, in line 9 of the algorithm, does not introduce session elements below **lastSessionTimestamp**, i.e., the elements that no longer belong to the suffix. The operation ensures this, because it only inserts elements with a timestamp above **lastTimestamp** and the session elements that have a timestamp above **lastTimestamp** belong to the suffix.

4.6.5.2 Read your Writes, Monotonic Reads, and Monotonic Writes

In this section, we are going to argue that the algorithm guarantees both Read Your Writes and Monotonic Reads after applying the Monotonic Writes transformation block. Recall the definition of an anomaly of Read Your Writes:

The get operation returns an older element inserted by the client before a more recent one, more precisely, there exist two elements x, y inserted over list L in the same client session, in this order(x then y), and a get returns S, the top of the list, and $y \notin S \wedge x \in S$.

To guarantee that this anomaly does not occur, the strategy used by Read Your Writes is to return a set (the truncated list of elements that are returned to the application) such that, when we project the elements from that list that belong to the current client session, we obtain a suffix of the sequence of elements inserted in that session. To preserve this, the Monotonic Writes transformation block cannot remove session elements that break the suffix. This is guaranteed because the Read Your Writes transformation block is executed before the Monotonic Writes transformation block, so the session suffix is complete, it has no gaps, and the Monotonic Writes algorithm will not remove session elements. The correct order is also maintained because the timestamps and the session counters increase monotonically in the session.

Finally, we argue that after applying Monotonic Writes transformation block we guarantee Monotonic Reads. Recall the definition of an anomaly of Monotonic Reads

When a client c issues two get operations that return sequences S_1 and S_2 (in that order) and the following property holds: $\exists x, y \in S_1 : S_1(x) \prec S_1(y) \wedge y \notin S_2 \wedge x \in S_2$, where $S_1(x) \prec S_1(y)$ means that element x appears in S_1 before y.

To guarantee that this anomaly does not occur, the strategy used by Monotonic Reads is to return a set (the truncated list of elements that are returned to the application) such that, when we project the elements from that list that were returned previously to the client, we obtain a suffix of the sequence of elements previously returned to the client. To preserve this, the Monotonic Writes transformation block cannot remove elements previous returned to the client larger or equal to **lastTimestamp**. To this end, its necessary to guarantee that the new elements inserted in *sl* do not create a gap below the timestamp of the oldest element from a session subsequence previously returned to the client, if this happens, the Monotonic Writes transformation block removes from *sl* the previously returned session subsequence. To avoid this situation, before

executing Monotonic Writes transformation block, in line 13 of Algorithm 7, we use the **localView** to know the previous subsequences returned to the client, and then we remove from *sl* the new elements of each previous subsequence older than the last element of each subsequence previously returned to the client.

4.6.5.3 Read your Writes, Monotonic Reads, Monotonic Writes, and Writes Follow Reads

In this section, we are going to argue that the algorithm guarantees Read Your Writes, Monotonic Reads, and Monotonic Writes after applying Writes Follow Reads transformation block. Recall the definition of an anomaly of Read Your Writes:

The get operation returns an older element inserted by the client before a more recent one, more precisely, there exist two elements x, y inserted over list L in the same client session, in this order(x then y), and a get returns S, the top of the list, and $y \notin S \land x \in S$.

To guarantee that this anomaly does not occur, the strategy used by Read Your Writes is to return a set (the truncated list of elements that are returned to the application) such that, when we project the elements from that list that belong to the current client session, we obtain a suffix of the sequence of elements inserted in that session. To preserve this, the Writes Follow Reads transformation block has to return a suffix of the session elements. It is necessary that the Writes Follow Read transformation block does not remove session elements that break the suffix. To guarantee this, we need to ensure that the dependencies of each session element until the last element of the truncated list returned to the client, are in *sl*. To this end, the elements stored in **localView** are included in *sl*, because the **localView** contains the elements previously returned to the client, which are the dependencies of the elements inserted in the session. For that, we reuse the Monotonic Reads transformation block code, which adds the elements in the **localView** to *sl*, and removes the old dependencies, i.e., the elements that do not belong to the truncated list returned to the client. This is done before executing the Writes Follow Reads transformation block

which guarantees that a suffix of the session elements and their dependencies are included in *sl* and returned.

Next, we argue that after applying Writes Follow Reads transformation block we guarantee Monotonic Reads. Recall the definition of an anomaly of Monotonic Reads:

When a client c issues two get operations that return sequences S_1 and S_2 (in that order) and the following property holds: $\exists x, y \in S_1 : S_1(x) \prec S_1(y) \wedge y \notin S_2 \wedge x \in S_2$, where $S_1(x) \prec S_1(y)$ means that element x appears in S_1 before y.

To guarantee that this anomaly does not occur, the strategy used by Monotonic Reads is to return a set (the truncated list of elements that are returned to the application) such that, when we project the elements from that list that were returned previously to the client, we obtain a suffix of the sequence of elements previously returned to the client. To preserve this, the Writes Follow Reads transformation block has to preserve the suffix of the sequence of elements previously returned to the client. To this end, the Writes Follow Reads transformation block cannot remove elements that break the suffix. This is automatically guaranteed, because the previous elements returned to the client are in the **localView** with all dependencies until **lastTimestamp**, and were included in *sl* by the Monotonic Reads transformation block.

Finally, we argue that after applying the Writes Follow Reads transformation block we guarantee Monotonic Write. Recall the definition of an anomaly of Monotonic Writes:

The get operation returns a subsequence of elements from a session in a different order they were issued or with gaps, more precisely, given a sequence of writes W in the same session, and a sequence S returned by a read: $(\exists x, y, z \in W : W(x) \prec W(y) \prec W(z) \wedge x \in S \wedge y \notin S \wedge z \in S) \vee (\exists x, y \in W : W(x) \prec W(y) \wedge S(y) \prec S(x))$.

To guarantee that this anomaly does not occur, the strategy used by Monotonic Writes is to return a set (the truncated list of elements that are returned to the application) such that, when we project each session subsequence they are ordered by insertion order and without gaps. To preserve this, the Writes Follow Reads transformation block cannot remove elements from a session subsequence that causes a gap in that subsequence. This again is already guaranteed,

because the dependencies of a session element include the dependencies of the previous session element in the session sub-sequence. Therefore, when we apply the Writes Follow Reads transformation block, we know the dependencies, directly through the dependencies list associated to each element or through the **cutTimestamp**, that defines that all elements below this timestamp are also dependencies. Since the Writes Follow Reads transformation block guarantees that the sublist returned to the client contains for each element, the dependencies with a timestamp larger or equal to the timestamp of the oldest element that is going to be returned, it is guaranteed that a gap in a session subsequence is not created. The order of each subsequence is also maintained, because the timestamps and the session counters increase monotonically in the session. Finally, the operation that was introduced in line 21 of Algorithm 7 to avoid the third corner case described in section 4.3.6, that removes all elements with a timestamp below a missing session element, is also safe, because the strategy employed to truncate the list to be returned to the client, maintains a suffix of a sublist that contains for each element, the dependencies with a timestamp larger or equal to the timestamp of the oldest element present in that sublist.

4.7 Comparison with Related Work

Here, we revisit the related work comparison in light of the contributions reported in this Chapter. In particular, we focus on a detailed contrast to the more closely related proposals found in the literature. The closest related work are the recent proposals that also target the use of a middleware layer that can mediate access to a storage system in order to upgrade the respective consistency guarantees.

In particular, Bailis et al. [16] proposed a system called bolt-on to offer causal consistency. There are two main distinctions between bolt-on and our proposal: first, we provide a fine-grained choice of which session guarantees the programmer intends the system to provide, and only pay a performance penalty associated with the cost of enforcing those guarantees. Second, they assume the underlying system offers a general read/write storage interface,

which gives significant more flexibility in terms of the system design than our proposal, which is restricted to the APIs provided by social networking services.

Bermbach et al. [20] also proposed a middleware to enforce consistency guarantees on top of data stores, namely, Amazon S3 [9], DynamoDB [32], or SimpleDB [7], in contrast to our focus on high level service APIs. They also do not provide to programmers a fine-grained choice to all session properties.

The last closest related work is the proposal from Brantner et al. [22], that proposes a middleware that provides atomic transactions and all session guarantees on top of Amazon S3. To this end, they use an external service to enforce the session guarantees, namely, the Simple Queuing System. This contrasts with our work because we do not use external services to guarantee the session properties, instead focusing on a shim layer that operates at the client side.

Finally, another important comparison to previous proposals is that they assume that the services do not impose rate limits to operations. If a limit is exceeded the application is blocked, this may happen when a client issues a get operation and the service misses an element. In this situation, an algorithm may need to do an extra request to obtain the element and the application can be blocked.

4.8 Summary

We have shown that it is possible to enforce different consistency properties, in particular session guarantees for applications that access online services through their public APIs. We do so without knowing the service architecture, and without assuming that the service itself provides any of these guarantees. Our solution relies on a thin Middleware layer that executes on the client side, and intercepts all interactions of the client with the online service. We have presented different algorithms to enforce each of the well known session guarantees. Furthermore, our algorithms follow a simple structure that allows to combine them easily. We have developed a prototype in Java that we used to evaluate our approach using two services: Facebook, and a geo-replicated

deployment of Redis. Our experiments show that we can enforce session guarantees with a modest overhead both in terms of user-perceived latency and communication with the services.

Chapter 5

Conclusions

In this thesis we have presented a measurement study of the consistency offered by the APIs of four online services. To this end, we started by identifying a set of anomalies that are not allowed by various consistency levels, and devised two tests that have the ability to expose these anomalies. Our measurement study, based on these tests, were conducted on Google+, Blogger, Facebook Feed, and Facebook Groups for an aggregate period of one month in each service. During the execution of our tests we detected several situations that increased the difficulty to test the services. Namely, the different requests rate limits imposed by the services, that made it impossible for our agents to do more requests for a long period of time. The analysis of the collected data from the tests showed the relatively frequent occurrence of most of the anomalies across all services except Blogger, which might suggest that the architecture used by this service enforces strong consistency. We also measured the divergence window between two agents and we found that in some services these are significantly shorter than in others, and in some situations the tests ended with the two agents observing divergent states of the system. Some of these results may be acceptable from the perspective of the users, but there are applications where this may be important, e.g., applications that are producing statistic information or need to do some synchronous action in different locations. This highlighted the need

for application developers to consider whether the intended semantics for their applications is compatible with these behaviors, and if not, to possibly write programs in a way that masks these anomalies. This study was published in DSN2016 [37].

With the results obtained from the measurement study we concluded that we can expect to find many consistency anomalies in the behavior of many online services, so our next step was to create a solution to enforce fine-grained consistency to applications that are using the services. To this end, we created a middleware that demonstrated the feasibility of enforcing different consistency properties, in particular session guarantees, for third party applications that access online services through their public APIs on the client side. We do so without explicit support from the service architecture, and without assuming that the service itself provides any of these guarantees. Again, we had to take into account the restrictions imposed by the service, namely, having a get operation that truncates the number of elements in a feed, the request rate limits, that restricts the number of extra requests that may be done from our middleware to obtain an element that is missing, and having no place reserved to store the metadata associated to an element (e.g., an entry in a data object). We believe this is something that makes sense to be included in the future by the services, to provide application developers the flexibility to associate extra information to the application data. Since most of the services already return metadata associated to a post, this appears to be an easy and useful feature to be supported by these services.

The middleware layer that we developed executes on the client side, and intercepts all interactions of the client with the online service, and avoids making extra requests when something is missing. We have presented four algorithms that enforce each of the well known session guarantees. Furthermore, our algorithms follow a structure that allows to combine them. We also have developed a prototype in Java that we used to evaluate our approach using two services: Facebook, and a geo-replicated deployment of Redis. Our experiments showed that we can enforce session guarantees with a modest overhead both in terms of user-perceived latency and communication with the centralized service. Note

that our middleware is not limited to be used by these services, it can be used by services that provide a read/write interface compatible with the middleware interface. This work was published in SRDS2017 [38].

5.1 Future work

While the results presented in this thesis provide the tools and results for a better understanding of the consistency anomalies that are prevalent in today's online services, while also proposing a client-side (generic) approach to enrich the consistency guarantees of such services, we believe that there are some interesting venues for future research:

The measurement study was made with four services: Facebook Feed, Facebook Group, Google+, and Blogger. Extending the study to more services like Instagram or Linkedin can enrich the results that were provided and give a broader view of the consistency guarantees effectively offered by these services.

Another important extension to our study, is to detect anomalies in internal components of service when the access to these components are available. In this case, instead of considering only black box tests we can apply our methodology to perform white-box testing to large-scale storage systems.

Our middleware layer enforces the four session guarantees, it might be relevant to ensure to application developers other consistency properties, properties that can be enforced on top of these service without a significant cost to applications, e.g., the guarantee that two clients do not diverge or the guarantee that the divergence window is bounded and does not affect availability.

We chose to support the main operations provided by these services, however some services provide other operations. Extending the middleware to support more operations, e.g., feed pagination, can be useful for application developers and a great challenge if we take into account the restrictions imposed by services in terms of number of API calls issued in limited time windows.

Bibliography

[1] D. Abadi. "Consistency tradeoffs in modern distributed database system design: CAP is only part of the story." In: *IEEE Computer* 45.2 (2012), pp. 37–42.

[2] D. Agrawal, A. El Abbadi, and R. C. Steinke. "Epidemic algorithms in replicated databases." In: *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM. Tucson, Arizona, USA, 1997, pp. 161–172.

[3] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. "Causal memory: Definitions, implementation, and programming." In: *Springer Distributed Computing* 9.1 (1995), pp. 37–49.

[4] B. F. C. et. al. "Benchmarking Cloud Serving Systems with YCSB." In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM. Indianapolis, Indiana, USA, 2010, pp. 143–154.

[5] S. Almeida, J. Leitão, and L. Rodrigues. "ChainReaction: a causal+ consistent datastore based on chain replication." In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. Prague, Czech Republic, 2013, pp. 85–98.

[6] *Amazon Elastic Compute Cloud (Amazon EC2)*. https://aws.amazon.com/ec2 (accessed Mar-2019).

[7] *Amazon Simple Database*. https://aws.amazon.com/simpledb (accessed Mar-2019).

[8] *Amazon Simple Queue Service*. https://aws.amazon.com/sqs/ (accessed Mar-2019).

[9] *Amazon Simple Storage Service*. https:/aws.amazon.com/s3 (accessed Mar-2019).

[10] E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. J. Wylie. "What consistency does your key-value store actually provide?" In: *Proceedings of the Sixth Workshop on Hot Topics in System Dependability*. Vancouver, BC, Canada, 2010, pp. 1–16.

[11] *Apache HBase*. https://hbase.apache.org/ (accessed Mar-2019).

[12] *Apache HTTP Server*. https://httpd.apache.org (accessed Mar-2019).

[13] H. Attiya, F. Ellen, and A. Morrison. "Limitations of highly-available eventually-consistent data stores." In: *IEEE Transactions on Parallel and Distributed Systems* 28.1 (2017), pp. 141–155.

[14] *Azure Databases*. https:/azure.microsoft.com/en-us/product-categories/databases (accessed Mar-2019).

[15] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. "Probabilistically bounded staleness for practical partial quorums." In: *Proceedings of the VLDB Endowment* 5.8 (2012), pp. 776–787.

[16] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. "Bolt-on causal consistency." In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. New York, USA, 2013, pp. 761–772.

[17] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. "Megastore: Providing scalable, highly available storage for interactive services." In: *Fifth Biennial Conference on Innovative Data Systems Research*. CA, USA, January, 2011.

[18] N. M. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. "PRACTI Replication." In: *Proceedings of the 3rd Conference on Networked Systems Design and Implementation*. USENIX. San Jose, California, USA, 2006, pp. 5–5.

[19] D. Bermbach and S. Tai. "Eventual Consistency: How Soon is Eventual? An Evaluation of Amazon S3's Consistency Behavior." In: *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*. ACM. Lisbon, Portugal, 2011, 1:1–1:6.

[20] D. Bermbach, J. Kuhlenkamp, B. Derre, M. Klems, and S. Tai. "A Middleware Guaranteeing Client-Centric Consistency on Top of Eventually Consistent Datastores." In: *Cloud Engineering (IC2E), 2013 IEEE International Conference on*. IEEE. San Francisco, CA, USA, 2013, pp. 114–123.

[21] *Blogger API*. `https://developers.google.com/blogger//` (accessed Mar-2019).

[22] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. "Building a Database on S3." In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. ACM. Vancouver, Canada, 2008, pp. 251–264.

[23] M. Bravo, L. Rodrigues, and P. Van Roy. "Saturn: A distributed metadata service for causal consistency." In: *Proceedings of the Twelfth European Conference on Computer Systems*. ACM. Belgrade, Serbia, 2017, pp. 111–126.

[24] E. A. Brewer. "Towards robust distributed systems." In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. Portland, Oregon, USA, 2000.

[25] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, et al. "TAO: Facebook's Distributed Data Store for the Social Graph." In: *2013 USENIX Annual Technical Conference*. San Jose, CA, USA, 2013, pp. 49–60.

[26] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. "From session causality to causal consistency." In: *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. Coruna, Spain, 2004, pp. 152–158.

[27] *Cassandra Database*. https://cassandra.apache.org (accessed Mar-2019).

[28] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. "PNUTS: Yahoo!'s hosted data serving platform." In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1277–1288.

[29] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. "Spanner: Google's globally distributed database." In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[30] F. Cristian. "Probabilistic clock synchronization." In: *Springer Distributed computing* 3.3 (1989), pp. 146–158.

[31] S. B. Davidson, H. Garcia-Molina, and D. Skeen. "Consistency in a partitioned network: a survey." In: *ACM Computing Surveys (CSUR)* 17.3 (1985), pp. 341–370.

[32] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: amazon's highly available key-value store." In: *ACM SIGOPS Operating Systems Review* 41.6 (2007), pp. 205–220.

[33] *Facebook API*. https://developers.facebook.com/docs/apis-and-sdks (accessed Mar-2019).

[34] *Facebook API - groups feed*. https://developers.facebook.com/docs/graph-api/reference/v3.2/group/feed// (accessed Mar-2019).

[35] *Facebook API - news feed*. https://developers.facebook.com/docs/graph-api/reference/v3.0/user/home// (accessed Mar-2019).

[36] *Facebook Service*. https://www.facebook.com (accessed Mar-2019).

[37] F. Freitas, J. Leitao, N. Preguiça, and R. Rodrigues. "Characterizing the Consistency of Online Services (Practical Experience Report)." In: *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*. IEEE. Toulouse, France, 2016, pp. 638–645.

[38] F. Freitas, J. Leitão, N. Preguiça, and R. Rodrigues. "Fine-Grained Consistency Upgrades for Online Services." In: *Reliable Distributed Systems (SRDS), 2017 IEEE 36th Symposium on*. IEEE. Hong Kong, 2017, pp. 1–10.

[39] S. Gilbert and N. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." In: *ACM Sigact News* 33.2 (2002), pp. 51–59.

[40] *Google API*. `https://developers.google.com/` (accessed Mar-2019).

[41] *Google App Engine*. `https://cloud.google.com/appengine` (accessed Mar-2019).

[42] M. P. Herlihy and J. M. Wing. "Linearizability: A correctness condition for concurrent objects." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492.

[43] *Jedis Redis Library*. `https://github.com/xetorthio/jedis` (accessed Mar-2019).

[44] M. E. Khan, F. Khan, et al. "A comparative study of white box, black box and grey box testing techniques." In: *Int. J. Adv. Comput. Sci. Appl* 3.6 (2012).

[45] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. "Providing high availability using lazy replication." In: *ACM Transactions on Computer Systems (TOCS)* 10.4 (1992), pp. 360–391.

[46] A. Lakshman and P. Malik. "Cassandra: a decentralized structured storage system." In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[47] L. Lamport. "Time, clocks, and the ordering of events in a distributed system." In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[48] L. Lamport. "On interprocess communication, Part 1: basic formalism, Part II: algorithms." In: *Distributed Computing. v1 i2* (1986), pp. 77–101.

[49] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS." In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. Cascais, Portugal, 2011, pp. 401–416.

[50] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. "Existential consistency: Measuring and understanding consistency at facebook." In: *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM. Monterey, California, 2015, pp. 295–310.

[51] P. Mahajan, L. Alvisi, M. Dahlin, et al. "Consistency, availability, and convergence." In: *University of Texas at Austin Tech Report* 11 (2011), p. 158.

[52] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. "Low-latency multi-datacenter databases using replicated commit." In: *Proceedings of the VLDB Endowment* 6.9 (2013), pp. 661–672.

[53] P. Mockapetris. *RFC1035: Domain names—implementation and specification*. 1987.

[54] *MySQL Database*. `https://www.mysql.com` (accessed Mar-2019).

[55] *Network Time Protocol(NTP)*. `http:http://www.ntp.org//` (accessed Mar-2019).

[56] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. "Detection of mutual inconsistency in distributed systems." In: *IEEE transactions on Software Engineering* (1983), pp. 240–247.

[57] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. "Flexible Update Propagation for Weakly Consistent Replication." In: *SIGOPS Operating Systems Review* 31 (1997), pp. 288–301.

[58] *Redis storage*. `https://redis.io/` (accessed Mar-2019).

[59] *REST Facebook Library*. `https://restfc.com` (accessed Mar-2019).

[60] Y. Saito and M. Shapiro. "Optimistic replication." In: *ACM Computing Surveys (CSUR)* 37.1 (2005), pp. 42–81.

[61] *SQLServer Database*. https://www.microsoft.com/en-us/sql-server/sql-server-2017 (accessed Mar-2019).

[62] A. S. Tanenbaum and M. Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[63] K. Tangwongsan, S. Tirthapura, and K.-L. Wu. "Parallel streaming frequency-based aggregates." In: *Proceedings of the 26th acm symposium on parallelism in algorithms and architectures*. ACM. Prague, Czech Republic, 2014, pp. 236–245.

[64] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. "Session guarantees for weakly consistent replicated data." In: *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*. IEEE. 1994, pp. 140–149.

[65] *Twitter API*. https://developer.twitter.com/ (accessed Mar-2019).

[66] *Twitter Service*. https://www.twitter.com (accessed Mar-2019).

[67] W. Vogels. "Eventually consistent." In: *Communications of the ACM* 52.1 (2009), pp. 40–44.

[68] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. "Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective." In: *Fifth Biennial Conference on Innovative Data Systems Research*. Vol. 11. Asilomar, CA, USA, 2011, pp. 134–143.

[69] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. "Bobtail: Avoiding Long Tails in the Cloud." In: *10th USENIX Symposium on Networked Systems Design and Implementation*. Vol. 13. Lombard, IL, 2013, pp. 329–342.

[70] H. Yu and A. Vahdat. "Design and evaluation of a continuous consistency model for replicated services." In: *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation-Volume 4*. USENIX. San Diego, California, 2000, pp. 305–318.

[71] H. Yu and A. Vahdat. "Design and Evaluation of a Conit-based Continuous Consistency Model for Replicated Services." In: *ACM Transactions on Computer Systems* 20.3 (2002), pp. 239–282.

[72] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. "Write fast, read in the past: Causal consistency for client-side applications." In: *Proceedings of the 16th Annual Middleware Conference*. ACM. Trento, Italy, 2015, pp. 75–87.

[73] K. Zellag and B. Kemme. "How consistent is your cloud application?" In: *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM. San Jose, California, 2012, 6:1–6:14.