Variable data consistency in Parse Server

André Santos Tavares

Thesis to obtain the Master of Science Degree in
Information Systems and Computer Engineering

Supervisors: Prof. Rodrigo Seromenho Miragaia Rodrigues

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Rodrigo Seromenho Miragaia Rodrigues
Members of the Committee: Prof. João Coelho Garcia

November 2017
Acknowledgments

I would like to thank my parents and grandmother for supporting and encouraging me to finish with success my studies, specifically my Master’s degree, over all these years.

I would like to thank all my friends and colleagues that helped me grow as a person.

I would also like to acknowledge my dissertation supervisors Prof. Rodrigo Rodrigues and Postdoctor Subhajit Sidhanta for their insight, support and sharing of knowledge that has made this Thesis possible.

Last but not least, I would like to thank the love of my life, Filipa Monteiro, who helped and motivated me to conclude this thesis.

To each and every one of you – Thank you.
Modern applications resort to specialized backend storage systems to store their persistent state, such as user accounts, shared content, documents and purchases. Replication, not only across various server replicas but also on the mobile device itself, is widely adopted by these backends in order to improve the reliability and performance. Although maintaining the strong consistency among replicas can guarantee the correctness of application behaviors, it will affect the application performance at the same time because there is a well-known trade-off between consistency and performance. The documentation of Parse Server, our target backend, does not clarify the level of consistency it provides. We further investigated on this subject and we came to the conclusion that the consistency guarantees of Parse Server are dependent on how some parameters, that have impact on the consistency of its storage system, MongoDB, are being set. Unfortunately, there is no information about that configuration. We also found that tuning these parameters might not guarantee that some consistency anomalies are averted. To improve on these guarantees, we propose a solution which allows the developers, of Parse apps, to choose the consistency guarantees they require, per object, and offer a new consistency model, provided by ZooKeeper, which works as a synchronization layer between Parse Server and MongoDB. In this context, ZooKeeper is considered the “Source of truth” regarding object versions. This solution enables Parse Server to provide Sequential consistency on its objects, if developers require it. We implemented and evaluated our proposed solution in order to measure how it impacts on the latency of Parse requests and on the throughput of Parse Server. The impact in certain cases is significant, but that is the price developers have to pay in order to achieve a higher level of data consistency on their
apps.

Keywords

Distributed Systems; Data replication; Data Consistency; Sequential consistency; ZooKeeper; Parse Server; MongoDB
Resumo

As aplicações de hoje em dia recorrem a sistemas de armazenamento backend, para armazenarem o seu estado persistente, como por exemplo contas de utilizadores, conteúdos compartilhados, documentos e compras. A replicação dos dados, não apenas em vários servidores réplica, mas também no próprio dispositivo móvel, é amplamente adotada por esses backends para melhorar a confiabilidade e o desempenho. Embora a manutenção da consistência forte entre as réplicas possa garantir o correto funcionamento das aplicações, isso afetará o desempenho da aplicação ao mesmo tempo, pois existe um trade-off bem conhecido entre consistência e desempenho. A documentação do Parse Server, o nosso backend alvo, não esclarece o nível de consistência que fornece. Nós investigamos sobre este assunto e chegamos à conclusão de que as garantias de consistência que o Parse Server oferece dependem de como alguns parâmetros, que têm impacto na consistência de seu sistema de armazenamento, MongoDB, estão a ser definidos. Infelizmente, não há informações sobre essa configuração. Nós também descobrimos que ajustar esses parâmetros pode não garantir que algumas anomalias de consistência sejam evitadas. Para melhorar essas garantias, propomos uma solução que permite aos programadores, de aplicações Parse, escolher as garantias de consistência que necessitam, por objeto, e oferecer um novo modelo de consistência, fornecido pelo ZooKeeper, que funciona como uma camada de sincronização entre o Parse Server e o MongoDB. Neste contexto, o ZooKeeper é considerado a “Fonte de verdade” em relação às versões de objetos. Esta solução permite que o Parse Server forneça consistência sequencial nos seus objetos, se os programadores assim o exigirem. Nós implementamos e avaliamos a nossa proposta de solução para medir como ela afeta a latência dos pedidos ao Parse Server e também o throughput do Parse Server. O impacto em certos casos é significativo, mas esse é o preço que os programadores têm que pagar para alcançar um nível maior de consistência.
de dados nas suas aplicações.

**Palavras Chave**

Sistemas distribuídos; Replicação de dados; Consistência de dados; Consistência sequencial; ZooKeeper; Parse Server; MongoDB
# Contents

1 Introduction .......................................................... 1
   1.1 Problem ........................................................... 4
   1.2 Goals .............................................................. 4
   1.3 Contributions ..................................................... 5
   1.4 Thesis Outline .................................................... 5

2 Related Work .......................................................... 7
   2.1 Backend as a Service (Backend as a Service (BaaS)) ..................... 9
   2.2 Parse Server ....................................................... 10
   2.3 MongoDB .......................................................... 10
   2.4 Simba: Tunable End-to-End Data Consistency for Mobile Apps ............ 11
   2.5 Putting Consistency Back into Eventual Consistency (Explicit Consistency) ...... 12
   2.6 Customizable and Extensible Deployment for Mobile/Cloud Applications (Sapphire) ... 12
   2.7 HyperDex: A Distributed, Searchable Key-Value Store ...................... 13
   2.8 ZooKeeper: Wait-free coordination for Internet-scale systems ............... 14
   2.9 Extensible Distributed Coordination (Extensible ZooKeeper) ............... 15
   2.10 Discussion ........................................................ 15
       2.10.1 Apply a novel consistency model .................................. 16
       2.10.2 Reusing an existing system with multi-level consistency ............. 16
       2.10.3 Replace MongoDB with Hyperdex ................................... 17
       2.10.4 Zookeeper as "source of truth" ................................... 17

3 Solution ................................................................. 19
   3.1 Decision ............................................................ 21
   3.2 Architecture ....................................................... 22
   3.3 Design .............................................................. 23

4 Implementation ......................................................... 25
   4.1 Changes to Parse Server .......................................... 27
       4.1.1 Communication with Zookeeper ................................... 27
# List of Figures

2.1 Client-Server applications: frontend & backend (from [1]) ........................................ 9  
2.2 Diagram of default routing of reads and writes to the primary (from [2]) ....................... 11  
2.3 The ensemble of ZooKeeper (from [3]) ............................................................................. 14  
3.1 Architecture Overview: Parse Server + ZooKeeper ............................................................ 22  
5.1 Default consistency vs Sequential consistency ................................................................. 37  
5.2 Default vs Sequential w/o inconsistencies vs Sequential w/ inconsistencies .................. 38  
5.3 Throughput when sending GET requests .......................................................................... 39  
5.4 Throughput when sending PUT requests .......................................................................... 39
List of Algorithms

1 POST ................................................................. 28
2 GET ................................................................. 29
3 PUT ................................................................. 30
4 DELETE ............................................................. 30
Acronyms

IaaS Infrastructure as a Service
BaaS Backend as a Service
EZK Extensible ZooKeeper
RPC Remote Procedure Call
API Application Programming Interface
SDK Software Development Kit
CRDT Conflict-free replicated data type
Introduction

Contents

1.1 Problem ......................................................... 4
1.2 Goals .......................................................... 4
1.3 Contributions .................................................. 5
1.4 Thesis Outline ................................................ 5
Most modern applications store data and interact with other services on the internet. For that purpose, these apps require complementary server-side platforms, often called backend, to store user accounts, shared content, documents, purchases, etc. These days, there are very few applications that do not require a backend at all. Regarding the choice of a backend, cloud services are emerging as the favorite backend option for all types of application requirements. These are a flexible and affordable solution for most applications since it allows companies to avoid or minimize upfront IT infrastructure costs, while offering easy scale up as computing needs increase and then scale down when demands decrease. For these reasons, hundreds of thousands of developers use cloud services for their apps. While many companies still manage their own backend platforms and deploy them on virtual machines running in the cloud using Infrastructure as a Service (IaaS) such as EC2 [4], another option that is now available to the developers of mobile applications is BaaS (also known as MBaaS, i.e. Mobile Backend as a Service). Services such as Amazon S3 [5] and Microsoft Azure [6] can be a great option if the developers do not want to manage and deploy their own backend. Application Programming Interface (API)s and Software Development Kit (SDK)s offered by BaaS providers allow programmers to easily connect to cloud servers and have functionalities such as user authentication, management and push notifications, and some providers even offer integrated analytics to monitor apps [7]. These services can offer many advantages, such as eliminate the hassle of configuring cloud servers, they are easy to use through widely used APIs (such as REST APIs), and allow developers to focus on the core functionality of the app, thus reducing the development effort. Parse Server [8] is our target BaaS. It is a widely known application development platform that offers a set of tools that make it easier for programmers to focus on front-end development, taking care of things like authentication, data storage, and push notifications. Currently, Parse Server is being used by hundreds of thousands of apps [9] [10].

Data replication is widely adopted by these backends (including Parse Server) in order to improve reliability and performance. This replication happens both at the service end, where multiple server replicas provide high availability and fault tolerance, and also by creating cached copies of the data at the mobile device, for improved performance and disconnected operation. Replication can cause consistency problems between multiple copies of the data. The multiple copies are strongly consistent if a read operation returns the same value from all copies and a write operation is a single atomic operation (transaction), which updates all copies before any other operation takes place. Though maintaining strong consistency of the replicated data can guarantee the correctness of application behavior, it will affect the application performance at the same time because there is a well-known trade-off between consistency and performance, explained by the need that strong consistency protocols have to synchronize replicas before issuing a reply to the clients [11].
1.1 Problem

When trying to understand where Parse Server is placed within this trade-off, we were faced with the lack of documentation on this subject. After a deeper study, we found that Parse Server offers the consistency guarantees that its storage system, MongoDB [12], offers. So, we consulted MongoDB’s documentation and we found that the consistency guarantees that MongoDB offers are dependent on how Parse Server is using MongoDB and, in particular, how Parse Server is setting some MongoDB parameters (such as write concern, read concern and read preference), which have an impact on the consistency of its data. Knowing that, we searched for information about that usage but we could not find any. Due to this uncertainty about how MongoDB is being used, we avoided rely on the assumption that Parse Server offers any specific consistency model, other than the fact that reads return a previously written values. Later, during the development of this thesis, we found a very interesting article that made us suspect that, even if they were setting or allowing the developers to set all those consistency related parameters from MongoDB, there are no guarantees that MongoDB provides strong consistency on its documents, particularly during server crashes and asynchrony. The article is from the website JEPSEN [13], which is a quite active and widely accepted website run by Kyle Kingsbury. JEPSEN tests clusters of datastore-type systems by firing data at them, introducing internal partitions and failures, and comparing what the system told the clients that it had stored, with what it had actually persisted, pushing vendors to make accurate claims and test their software rigorously. In this article [14] from their website, they tested different open source releases of MongoDB and report whether certain anomalies like lost updates, dirty reads or stale reads are averted in the current release. Looking at the end of this article, it is clear that the last version of MongoDB seems to avert these consistency anomalies only when the parameters related to consistency are configured in a specific way. While this is a possible route worth exploring, it was not available at the beginning of this thesis and therefore we chose a different path.

1.2 Goals

The data consistency guarantees offered by Parse Server are unknown and that is a problem because it can compromise the correctness of applications, which ultimately means that the user experience can be negatively affected. Since we cannot expect any consistency model from either Parse Server or the underlying storage systems that it uses, we aim to improve Parse Server consistency guarantees by extending it to support a more principled approach towards providing consistent access to the data it stores. Specifically, we want to build on recent proposals for multi-level consistency [15], which allow the developer (or an automated tool on their behalf [16]) to associate an appropriate consistency level to each Parse object. Thus, we think that exposing consistency to developers this way could be a very interesting design decision because it allows them to be more aware of consistency issues,
gives them more control, and allows for consistency to become a first order concern in the programming of applications. In order for our extension have greater probability of being accepted by the Parse Server community, we will have to make as few and local changes on Parse Server and its SDKs as possible, to ease the maintenance, and the adaptation of existing Parse applications will have to be as easy as possible, in order to developers quickly start using it. Also, the impact on performance, meaning on latency and throughput of Parse Server, will have to be minimal.

1.3 Contributions

We aim to contribute in several ways. We analyzed many potential solutions to provide different consistency models on Parse objects, by understanding their advantages and limitations. We implemented a solution to improve the consistency guarantees of Parse Server which allows the developers to choose the consistency guarantees they require, per object. We evaluated our proposed solution in order to measure how it impacts on the latency and throughput of Parse requests.

1.4 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 (Related Work) provides background information that is needed to better understand the problem this thesis tries to solve, and presents work from different authors, which helped to find a solution to the presented problem, and discusses about solution alternatives to our problem. Chapter 3 (Solution) presents the decision of a solution, explaining its architecture and design; Chapter 4 (Implementation) presents the implementation process, the solution's design in detail and all the tools used to create it; Chapter 5 (Evaluation) shows the results from the evaluation process in order to get a good insight about the quality of the solution; Chapter 6 (Conclusion) concludes this thesis with its results and contributions, and states all the work that needs to be done in order to the solution be as complete as possible, as well as some extra ideas to implement in the future, in order to improve it, in case of the solution be adopted.
Related Work

Contents

2.1 Backend as a Service (BaaS) ........................................... 9
2.2 Parse Server ............................................................ 10
2.3 MongoDB ............................................................... 10
2.4 Simba: Tunable End-to-End Data Consistency for Mobile Apps .... 11
2.5 Putting Consistency Back into Eventual Consistency (Explicit Consistency) . . 12
2.6 Customizable and Extensible Deployment for Mobile/Cloud Applications (Sapphire) 12
2.7 HyperDex: A Distributed, Searchable Key-Value Store .................. 13
2.8 ZooKeeper: Wait-free coordination for Internet-scale systems ............ 14
2.9 Extensible Distributed Coordination (Extensible ZooKeeper) ............... 15
2.10 Discussion .............................................................. 15
In this chapter we present Parse Server, the backend that we want to improve. We also present and discuss our research on previous work related to providing different levels of consistency, in order to have a good insight about the possibilities and existing solutions for the problem we are trying to solve.

## 2.1 Backend as a Service (BaaS)

A large group of today’s applications can be placed in the category of "client-server applications". These are applications that consist of a client side, called the frontend, and a server side, called the backend, as showed on Figure 2.1. The frontend is, ultimately, what the user sees on the screen, whether it is the screen of a mobile device or a computer monitor. The frontend is responsible for guiding the user through the workflows established by the application, performing data posting and allowing the visualization of request results. The backend consists of two main components: application logic and data processing and management. The data processing and management involves users, persistence of data, post-subscription messages, e-mail delivery, geolocation data, files and media streams, etc. The application logic is mostly the workflow logic, the application specific coordination of domain and infrastructure components according to the requirements of that particular application. The separation of the application logic from the rest of the data processing and management tasks is an important design decision because, without the application logic, the rest of the server-side code becomes generic and can be used by any client-server application. As a result, these functions can be grouped as reusable services and, by combining them, can be consumed through a Backend as a Service (BaaS).

![Figure 2.1: Client-Server applications: frontend & backend (from [1])](image)
2.2 Parse Server

Parse is the target BaaS of this thesis. Parse is a widely known application development platform, which was acquired by Facebook in 2013. Recently, the development of Parse has been discontinued because Facebook’s strategy changed and they are now focusing on another platform called Messenger and so they moved the Parse team to work on that product. After Facebook made that decision, they released an open source version of Parse called Parse Server [8], which allows programmers to run most of the Parse API from their own Node.js server. Since then, Parse Server has been growing thanks to the community that uses it. Using Parse Server, programmers work with application-level concepts like user accounts and push notifications rather than technology-focused concepts like databases and socket I/O. Parse Server uses MongoDB [12] as its storage database. Parse was shutdown on 30 Jan of 2017 and, by that time, it was still being used by hundreds of thousands of apps [9] [10].

2.3 MongoDB

Parse Server uses MongoDB to store its data. MongoDB (version 3.4) offers a total order on write operations when there are no server crashes by default: reads and writes are issued to the primary member of a replica set, as showed in Figure 2.2. Applications can optionally read from secondary replicas in order to increase availability. In that case, when there are no server crashes, the system offers a consistency level that has been termed timeline consistency [17], which means that reads can be temporarily slightly out of date. There are three parameters that can be set in order to tune the data consistency guarantees that the programmer wants MongoDB to provide. These are Write Concern, Read Concern and Read Preference.

Write Concern describes the level of acknowledgment requested from MongoDB for write operations to a standalone mongod, to replica sets or to sharded clusters. The default value of the write concern is “1”. Write concern set to “majority” means that it is requested acknowledgment from the majority of the mongod instances.

Read Concern query option for replica sets and replica set with sharded clusters determines which data to return from a query. For example, using “linearizable”, the query returns data that reflects all successful writes issued with a write concern of “majority” and acknowledged prior to the start of the read operation. Regardless of the read concern level, the most recent data on a node may not reflect the most recent version of the data in the system. The default value is “local”, where the query returns the instance’s most recent data. This provides no guarantee that the returned data has been written to a majority of the replica set members (i.e., subsequent reads may return a prior write).

Read preference describes how MongoDB clients route read operations to the members of a replica set. By default, an application directs its read operations to the primary member in a replica set. Modes
other than “primary” may return stale data because, with asynchronous replication, data in the secondary may not reflect the most recent write operations.

![Diagram of default routing of reads and writes to the primary](image)

**Figure 2.2:** Diagram of default routing of reads and writes to the primary (from [2])

### 2.4 Simba: Tunable End-to-End Data Consistency for Mobile Apps

Simba [18] presents sTable, a high-level programming abstraction, design for developing mobile applications connected to the cloud. It provides tunable end-to-end data consistency, with a unified data model for table and object data, and atomicity over coarse-grained and inter-dependent data, meaning data granularity relevant in the application. For example, data related to an email. To demonstrate the utility and practicality of sTables, they created Simba, a data synchronization service. Simba treats a table as the unit of consistency specification and a row as the unit of atomicity preservation. Each sTable has associated one of three consistency levels: Sequential, Causal and Eventual. Programmers define the consistency level per table, for example, a table of images, where each row can be viewed as an application-level object, composed by the collection of objects related to a specific image, and each row is treated atomically in every read and write.
2.5 Putting Consistency Back into Eventual Consistency (Explicit Consistency)

The authors present Explicit Consistency [19], a consistency model that ensures application correctness, centered on its semantics, and not on the order of operations. It offers eventual consistency by requiring application-specific invariants, which are defined by the programmers. Explicit Consistency identifies which operations could not be safe to execute concurrently, and allows the programmers to choose between violation-avoidance or invariant-repair techniques. Violation-avoidance is achieved by relying on a reservation system that moves replica coordination away from operation execution. Invariant-repair, in turn, allows operations to execute without restriction, and restores invariants by applying a repair operation to the database state. The system is free to reorder execution of operations at different replicas, provided that the specified invariants are maintained. They showed that it is possible to implement explicit consistency while mostly avoid cross-data-center coordination. First, they use static analysis to infer which operations can be safely executed without coordination. Second, for the remaining operations, they provide to the programmer a choice of either invariant-repair or violation-avoidance techniques. Finally, the application code is instrumented with the appropriate calls to their middleware library of objects that repair invariants automatically, using techniques similar to the ones employed by Conflict-free replicated data type (CRDT)s [20]. Programmers may extend this library in order to support additional types of invariants.

2.6 Customizable and Extensible Deployment for Mobile/Cloud Applications (Sapphire)

Sapphire [21] is a distributed programming platform that simplifies the programming of mobile and cloud applications by separating the application logic from the deployment logic. Sapphire’s key design feature is its distributed runtime system, which supports a flexible and extensible deployment layer for solving complex distributed tasks, such as data consistency, fault-tolerance, code-offloading and caching. Rather than writing distributed systems code, programmers choose from a list of deployment managers that extend Sapphire’s kernel to meet their applications’ deployment requirements. They can also create custom deployment managers. With this flexibility, programmers can quickly switch deployment solutions to respond to environment or requirement changes, or simply to test and compare alternatives during development.

A Sapphire application is composed by a set of Sapphire Objects with a corresponding set of Deployment Managers (DMs), and a Deployment Kernel (DK). Sapphire Object is the granularity of distribution and decomposition in Sapphire. Deployment Managers, which are extensions to the Deployment Kernel,
support complex management tasks, such as data consistency, controls over placement and Remote Procedure Call (RPC) semantics, fault-tolerance, load balancing and scaling, code-offloading, and peer-to-peer deployment. Each Sapphire Object can optionally have a Deployment Manager attached. The Deployment Kernel manages and keeps track of the location of Sapphire Objects, it supports communications between objects (RPC), low-level replica support, and has services to simplify the writing and execution of Deployment Managers.

2.7 HyperDex: A Distributed, Searchable Key-Value Store

HyperDex [22] is a high-performance, scalable, consistent and distributed key-value store that combines strong consistency guarantees with high availability in the presence of failures and partitions affecting up to a certain number of servers. In addition, HyperDex provides an efficient search primitive for querying objects through their secondary attributes. It achieves this last functionality through hyper-space hashing, in which objects with multiple attributes are mapped into a multidimensional hyperspace. This mapping leads to efficient implementations for key-based retrieval, partially-specified searches and queries within a range. HyperDex uses a replication protocol called value-dependent chaining, to simultaneously achieve fault tolerance, high performance and strong consistency. Regarding the consistency of the keys, HyperDex guarantees that all operations on a specific key (e.g., get and put) are linearizable [23] with all operations on all keys. Concerning consistency when searching, HyperDex guarantees that a search will return all objects that were committed at the time of search, meaning that an application whose put succeeds is guaranteed to see the object in a future search. When concurrent updates are occurring, a search may return either the committed version or the newly updated version of an object which matches the search query. Thus, HyperDex provides the strongest form of consistency for write operations, and a conservative and predictable consistency guarantees for read operations. An evaluation of the full system shows that HyperDex is 12-13x faster than Cassandra and MongoDB for finding partially specified objects. Additionally, HyperDex achieves 2-4x higher throughput for gets and puts.

HyperDex has an API compatibility with MongoDB (section 2.3) called Mongo Veneer [24]. That API makes it possible to seamlessly switch from MongoDB to HyperDex (i.e., can act as a stand-in for MongoDB). The authors claim that HyperDex combined with Mongo Veneer is 1-4x faster than MongoDB itself. The Mongo veneer automatically creates and manages the necessary data spaces, and convert MongoDB queries and updates to HyperDex calls. Also, it supports many of the operations and operators supported by MongoDB. The Mongo veneer provides coverage of many basic operations within MongoDB, but certain features are not available. HyperDex provides strong consistency in all configurations, and so, every object retrieval always returns the most recent written object. There is no need to
tune journal parameters, write concern, read concern, write timeout or the dozen other parameters that
MongoDB requires to be set correctly.

2.8 ZooKeeper: Wait-free coordination for Internet-scale systems

ZooKeeper [25] is a replicated synchronization service for coordinating processes of distributed applications. It is used for maintaining information about configuration, naming, provide distributed synchronization and group services. The interface exposed by ZooKeeper has wait-free data objects, organized hierarchically as in file systems, with an event-driven mechanism, similar to cache invalidations of distributed file systems, in order to provide a simple, yet powerful coordination service. It is robust, since, as showed in Figure 2.3, the persisted data is distributed between multiple nodes (this set of nodes is called an “ensemble”) and one client connects to any of them (i.e., a specific “server”). As long as a strict majority of nodes are working, the ensemble of ZooKeeper nodes is alive. Thus, consensus, group management, and other related protocols are implemented by this service so that the applications do not need to implement them on their own.

![ZooKeeper Service](image)

**Figure 2.3:** The ensemble of ZooKeeper (from [3])

Regarding its data consistency guarantees, ZooKeeper provides a guarantee of FIFO execution of requests per client, and linearizability for all requests that write on ZooKeeper. Each time a client writes to the ensemble, a majority of nodes persist the information. These nodes include the server with which the client is connected, and also the leader, which is the server that contains the full state of Zookeeper. This means that each write makes the server up-to-date with the leader. To guarantee that write operations satisfy linearizability, ZooKeeper uses a leader-based atomic broadcast protocol [26].

Reads are concurrent since they are served by the specific server that the client is connected. However, the “view” of a client may be outdated, since the leader updates the corresponding server with a bounded but undefined delay. Both read and write operations are designed to be fast, though reads are faster than writes. In the following list are the consistency guarantees provided by ZooKeeper:

- **Sequential Consistency**: Writes from a client will be applied, in all replicas, by the order that they
were sent.

- **Atomicity**: writes either succeed or fail, meaning that there are no partial writes.

- **Single System Image**: A client will see the same state of ZooKeeper regardless of the server that it is connected.

- **Reliability**: Once a write to a node has been committed, it will persist until a client overwrites that node.

- **Timeliness**: The clients view of the system state is guaranteed to be up-to-date within a bounded interval on the order of tens of seconds.

### 2.9 Extensible Distributed Coordination (Extensible ZooKeeper)

The authors present a model [27] that allows clients to dynamically and securely extend a coordination service by introducing small portions of custom code, which are executed atomically at the server side and they also constrain such extensions in order to not degrade or disrupt the performance of the system. They applied it to ZooKeeper, and called it Extensible ZooKeeper (EZK). The modifications to ZooKeeper are the following: EZK introduces two methods for registering and deregistering extensions into the ZooKeeper. Internally, these methods are mapped to ZooKeeper operations which will create and delete sub-objects of the data object representing the extension manager, for example, `/em/ex` for an extension `ex`. In order to customize the behavior of the coordination service though extensions, the extension manager must be able to monitor and control the handling of operations. They meet this requirement by invoking the extension manager at the preprocessor stage of ZooKeeper, which allows the interception of requests issued by clients and redirect them to extensions. In conclusion, the main advantage of EZK is that it allows clients to batch multiple API calls in the same RPC while ensuring its atomic execution. This drastically differs from Zookeeper, in which each API requires a RPC. This property enables clients to execute complex operations with very low latency.

### 2.10 Discussion

In this section we discuss our research by presenting several solution alternatives and our analysis of their advantages and disadvantages.
2.10.1 Apply a novel consistency model

There are advantages and disadvantages in Explicit Consistency (from section 2.5) when compared to a consistency model that allows the programmer to choose which consistency guarantee to use for each operation, such as Lazy [28] and RedBlue [15]. The main advantage of using Explicit Consistency model is that the programmer only writes invariants and does not have to be concerned about consistency. A disadvantage is that it requires programmers to write these invariants, which can be error-prone, leading to wrong definition of invariants, or not completely specify all needed invariants, thus compromising application correctness. Also, in addition to writing invariants, programmers must choose between violation-avoidance and invariant-repair techniques. Therefore, Explicit consistency requires programmers to make some considerable changes (define invariants, violation-avoidance and invariant-repair techniques) on existing applications, which we want to avoid at most. Another disadvantage is that the programmer does not have any control over the consistency guarantees of each operation or object.

2.10.2 Reusing an existing system with multi-level consistency

Comparing Sapphire with Parse Server (sections 2.6 and 2.2), they differ in the following point: the Parse Server architecture combines the DMs with the DK, and so there is no personalized deployment logic for each Parse Object, which means that only one DM exists. Implementing this architecture on Parse Server, one could personalize the consistency of each Parse Object through DM’s but the library of DM’s offered by the Sapphire is short in terms of DMs related to consistency, only providing three DMs concerning Serializability, meaning that developers have to, not only be concerned about the application logic, but also in developing DMs in order to provide other consistency models. Sapphire, in terms of the interface for developing mobile applications, takes a clean slate approach and tries to develop an API that exposes, to the application developer, several aspects such as consistency, caching, replication and scalability. However, we are more interested in an approach that privileges the seamless adaptation of existing code, by avoiding changes on the Parse Server API as much as possible.

Despite the fact that sTables, from Simba (section 2.4), is a well define programming abstraction with flexibility concerning data consistency, adapting existing Parse Server applications to use sTables can be quite challenging because programmers have to change how data is organized and accessed, which will potentially induce considerable changes on existing applications. sTables enable the use of multiple consistency levels in the same app, but consistency per table could not be the most efficient method. It could happen that some operations on rows of tables with a stronger consistency level could be executed using weaker consistency models, so overall performance could potentially be sub-optimal. sTables use could also be error-prone because programmers could choose to apply a weaker consistency level on operations that require stronger consistency level, which might later cause inconsistencies.
2.10.3 Replace MongoDB with HyperDex

Replacing MongoDB with HyperDex could be a solution. HyperDex is a completely different database from MongoDB, but it has a compatible API (i.e. it can act as a stand-in for MongoDB), provides strong consistency in all configurations and the authors say that it is 1-4x faster than MongoDB itself [24]. Thus, this seems to be a quite valid alternative to be implemented on Parse Server. However, this API does not support some MongoDB features, and there is no available information about which ones and so we can not argue if they are important ones or not. Thus, if we would choose to use HyperDex, we would have to test it a lot and possibly improve HyperDex if needed. Another important aspect to consider is that HyperDex has a much smaller user base, is much more specialized and it seems both development on the product and commercial support has stopped. In contrast, MongoDB is one of the most popular databases and it is used by thousands of companies so it has the advantage of being maintained, available, supported and it has a very large community around it.

2.10.4 Zookeeper as “source of truth”

Implementing a synchronization layer between Parse Server and MongoDB is also a possible solution. Instead of relying on MongoDB consistency, Parse Server can use an external coordination system, such as ZooKeeper, that offers Sequential consistency guarantees, to validate the consistency of MongoDB objects, by consulting and comparing “version values”, that will be stored on both systems. Despite the fact that this extra “layer” can potential induce some delay on Parse requests, the impact on performance is expected to not be significant since ZooKeeper has great performance on writes and specially on reads, as stated in section 2.8, and also because the files that would be stored in it would be very small. Regarding Extensible ZooKeeper (from section 2.9), it could make sense to use it if our solution needs to call ZooKeeper many times, for each Parse request, since it would lower a lot the overall latency of each request, improving the overall performance of our solution.
Contents

3.1 Decision ................................................................. 21
3.2 Architecture ........................................................... 22
3.3 Design ................................................................. 23
This chapter is organized in the following manner: On section 3.1, we present our decision on the design of the solution; section 3.2 has the architecture of the proposed solution; Finally, on section 3.3, we explain the design of our proposed solution.

### 3.1 Decision

Our decision to strengthen the consistency guarantees of Parse Server was to use an external co-ordination service, ZooKeeper, to provide another consistency model to Parse Server. The developers of Parse apps can now decide if they want Parse Server to provide a clear consistency model on their objects, Sequential consistency. This choice is justified by the various advantages we presented in the section 2.10, which outweigh the disadvantages of the other alternatives. Specifically, because of the Sequential guarantees ZooKeeper provides, because of the ease to use it in the context of this problem, and because of its performance.

In terms of the interface that our new system offers, we decided to expose consistency to developers by enabling Parse Server to provide two consistency levels: the Default one, provided by MongoDB, and Sequential consistency, which corresponds to the level of consistency that ZooKeeper is configured to provide on each deployment. For the first consistency model, we call it “Default” because it is not possible to give a more precise name, since there is no information about the consistency guarantees provided by Parse Server.

The idea behind using ZooKeeper is the following: On each read and write, our code gets and compares version numbers, associated to the MongoDB objects, that are stored on both MongoDB and ZooKeeper. Thus, in the context of this problem, Zookeeper is considered the “Source of truth”, meaning that in every read and write on MongoDB objects, those operations are first validated by querying the most recent version of the target object, that is stored on ZooKeeper, in order to ensure Sequential consistency on the target object. To implement this extra consistency model, we have to make some changes on Parse Server to also call ZooKeeper and compare those object versions. This changes are minimal and, above all, the majority of the changes is confined to a single location in the source code. Specifically, we need to change the file that make calls to MongoDB.

Regarding the changes on the code from the client side, meaning on the SDKs code, we decided to only change on Parse SDK since that is sufficient to evaluate our proposed solution. We chose to modify the Parse Android SDK because it is used by most of the existing Parse applications. Thus, we made minor changes on the Parse Android SDK in order for the programmer specify the consistency model for each Parse object.
### 3.2 Architecture

In this section we present the architecture of our proposed solution, which gives an overview of the main blocks of the whole system. This high-level architecture is depicted in Figure 3.1. The key change to the original architecture of Parse Server is that now, Parse Server instances are not only connected to a MongoDB replica set but also to a ZooKeeper ensemble. The following list presents and describes the main components of the system:

- **Clients** interact with applications that are using the modified Parse SDKs to make requests to Parse Server instances
- **Load balancer** distributes the workloads across all Parse Server instances
- **Parse Server instances** are back-end servers that serve the clients
- **MongoDB replica set** is the storage system where Parse Server stores all users data
- **ZooKeeper ensemble** is a replicated synchronization service that keeps record of the most recent version of each object stored on MongoDB

![Figure 3.1: Architecture Overview: Parse Server + ZooKeeper](image-url)
3.3 Design

In this section we present the design of our proposed solution. In the context of our problem, ZooKeeper, as "source of truth", determines what is the most recent version of each object stored on MongoDB. Considering that, the main design idea is the following: If the programmer does not set the consistency for the target object or if it sets it to "default", then the Parse Server will behave as it was behaving before, that is running the original calls to MongoDB and no calls to ZooKeeper neither run additional code; If the programmer specifies that he/she wants Sequential consistency for the target object, then Parse Server will make calls to ZooKeeper and run additional code, which can be compare versions, generate hashes, etc, depending on the type of request. In our proposed design, a version of an object is represented by an hash of it. Instead of an hash, we could also use just a random number, but we decided to use an hash. We present next our main design idea, by describing the control flow of actions for each type of Parse request, when Sequential consistency is required (detailed pseudocode on chapter 4):

Create Object :
1. Store the hash on ZooKeeper;
2. Store the object + the hash on MongoDB;

Get Object :
1. Get the object from MongoDB;
2. Get the hash from ZooKeeper;
3. If the hashes match, then return the object;
4. Else, use the hash from ZooKeeper to get the correspondent object from MongoDB;

Update Object :
1. Get the object as previously explained;
2. Store a copy of it, using its hash as id;
3. Update the hash on ZooKeeper with a new one;
4. Update the object on MongoDB as normal, but also update the hash on it with a new one;

Delete Object :
1. Get the object from MongoDB;
2. For each old version value in it, delete the correspondent object from MongoDB;
3. Delete the hash from ZooKeeper;
4. Delete the object from MongoDB;
4

Implementation

Contents

4.1 Changes to Parse Server .............................................. 27
4.2 Changes to Parse Android SDK ..................................... 29
In this chapter we present the implementation of the solution proposed on previous chapter 3. In the context of our problem, ZooKeeper, as "source of truth", determines what is the most recent version of each object stored on MongoDB. Thus, there is a need to verify if the version (object) that MongoDB returns is the most recent. In order to verify that, we changed the methods from Parse Server’s code where MongoDB is called, to respond to the incoming REST requests: POST, GET, PUT and DELETE, according to the consistency required, specified by the client. We present next these changes to Parse Server and we also present the changes to the Parse Android SDK, to allow developers to specify the consistency guarantees they need. All the code is on the following github repository: https://github.com/astavares/Parse-Server-with-ZooKeeper. Note: The code needs to be refactored, specially the method `findOneAndUpdate` (PUT), since it is very hard to read. The other methods that we changed should not be difficult to read and understand, since they are very similar to the pseudocode that is presented next.

4.1 Changes to Parse Server

We made minor changes to Parse Server in order to extract the consistency parameter from the header of each REST request (POST, PUT, GET and DELETE) received from the client. That parameter defines the consistency required for the target object of a request. Major changes were made on a file called "MongoStorageAdapter.js", which is where the code that make calls to MongoDB is, to also call ZooKeeper and compare the data from both. The algorithms that we use to store and retrieve data are summarized in the pseudocode presented on the next sections of this chapter. Note that, despite the fact that Parse Server is written in JavaScript (Node.js) and this language is asynchronous, all the calls to ZooKeeper and MongoDB on the pseudocode presented next are synchronous due to how Parse Server is built. In particular, it uses JavaScript Promises [29] to chain asynchronous calls, making the whole algorithm synchronous. The code must be synchronous, meaning the calls and callbacks to MongoDB and Zookeeper must be ordered in a sequential manner, in order to compare the data from MongoDB and ZooKeeper.

4.1.1 Communication with Zookeeper

The first step to implement our proposed solution is to enable Parse Server to communicate with ZooKeeper. ZooKeeper only offers an official API binding for Java and C, and Parse Server is written in JavaScript (Node.js). Since the programmers of ZooKeeper released a ZooKeeper client written in Java, we first tried to reuse it by calling that Java code. For that purpose, we tried many different Node.js modules in order to spawn a process running that Java client, and then send input and get the output. We tried, spawn, a method from a native Node.js module called Child Process [30], which launches
a new Child Process with the given command. Thus, we tried to spawn a terminal and then run the Java client to later send commands and receive their responses. The responses were being received through an event listener instead of a through a callback, and that raised a problem with the Promises that Parse Server is using to make the code synchronous, because Promises only work with callbacks. To circumvent this problem, we experimented using a Node.js module that is an unofficial ZooKeeper API called node-zookeeper [31], which is implemented on top of the ZooKeeper API written in C, and it works. Thus, we decided to use this module in order to enable the communication between Parse Server and ZooKeeper.

4.1.2 POST method

When a client sends a POST request to create a Parse Object, if Sequential consistency is not needed, than it simply stores the object on MongoDB. If Sequential consistency is needed, a hash of the object will be stored on ZooKeeper and also on MongoDB, inside of the object. That hash will be the "version value" of that object, for later consistency checks. See the pseudo-code on Algorithm 1.

Algorithm 1: POST

```plaintext
Function createObject (className, object, consistency)
  if consistency == SEQUENTIAL then
    path = generatePath(className, objectId);
    hash = generateHash(object);
    storeOnZK(path, hash);
    object.hashs = [hash];
  storeOnMongo(object);
```

4.1.3 GET method

When a client sends a GET request to find a Parse object, it first finds the object on MongoDB and, if Sequential consistency is not needed, then it simply returns the object to the client. If Sequential consistency is needed, then it gets from ZooKeeper the hash correspondent to that object, which represents the most recent version of it. Then it gets the hash from the object retrieved from MongoDB. If the hashes do not match, meaning that there is an inconsistency, then it repeatedly tries to find on MongoDB the object using that hash has objectId, until it finds it or until timeout (optimistic read). Finally, it returns the object to the client. See the pseudo-code on Algorithm 2.
Algorithm 2: GET

1 Function findObject (className, objectId, consistency)
2     object = findOnMongo(className, objectId);
3     if consistency == SEQUENTIAL then
4         path = generatePath(className, object.id);
5         hashFromZK = getFromZK(path);
6         objectHash = oldObject.hashs[0];
7         if objectHash != hashFromZK then
8             repeat
9                 object = findOnMongo(className, hashFromZK);
10            until object != null OR timeout;
11         return object;

4.1.4 PUT method

When a client sends a PUT request to update a Parse object, if Sequential consistency is not needed, then it simply sends the updates directly to MongoDB and returns to the client. If Sequential consistency is needed, then it gets the Parse object as previously explained in section 4.1.3. Then, it gets from that object all of its hashes (versions) and checks if the size of that array of hashes is equal or greater than some maximum number of old versions. If it is, then it makes a call to MongoDB in order to asynchronous delete the object corespondent to the oldest hash on that array. Then, it updates the olderHashs array because that object no longer exists. After doing that garbage collect work, it applies the updates on the object, generates a new hash and adds it to the updates. Then it updates the hash on ZooKeeper with the new one. Finally, it sends the updates to MongoDB. See the pseudo-code on Algorithm 3.

4.1.5 DELETE method

When a client sends a DELETE request to delete a Parse object, if Sequential consistency is not needed, then it simply deletes the object from MongoDB. If Sequential consistency is needed, then our system only needs to remove all the information related to the object. To this end, it first gets the object from MongoDB, and for each hash in the array of old hashes of that object, it asynchronously deletes the correspondent objects, meaning that it does not wait for the callbacks to run the final steps, which are: delete the hash from ZooKeeper and the object from MongoDB.

4.2 Changes to Parse Android SDK

Parse server has nine client SDKs, all of which communicate with Parse Server through a REST API. We decided to only change on Parse SDK since that is sufficient to evaluate our proposed solution.
Algorithm 3: PUT

Function updateObject (className, objectId, updates, consistency)

1. if consistency == SEQUENTIAL then
2.     path = generatePath(className, objectId);
3.     oldObject = findObject(className, objectId, consistency);
4.     oldObjectHash = oldObject.hashs[0];
5.     oldObject.id = oldObjectHash;
6.     storeOnMongo(oldObject);
7.     olderHashs = oldObject.hashs;
8.     oldVersionToBeDeleted = getOldVersionToBeDeleted(olderHashs);
9.     if oldVersionToBeDeleted != null then
10.        asyncDeleteOnMongo(oldVersionToBeDeleted);
11.        olderHashs = updateOlderHashs(olderHashs);
12.        updatedObject = applyUpdates(oldObject, updates);
13.        newHash = generateHash(updatedObject);
14.        updates.hashs = concatenate(newHash, olderHashs);
15.        updateOnZK(path, newHash);
16.        updateOnMongo(className, objectId, updates);

Algorithm 4: DELETE

Function deleteObject (className, objectId, consistency)

1. if consistency == SEQUENTIAL then
2.     object = findOnMongo(className, objectId);
3.     foreach hash in object.hashs do
4.         asyncDeleteOnMongo(hash);
5.         deleteOnZK(path);
6.     deleteOnMongo(className, objectId);
and because our extension must first be accepted by the Parse Server community. We chose to modify the Parse Android SDK because it is used by most of the existing Parse applications. We made minor changes on the Parse Android SDK in order for the programmer specify the consistency model for each Parse object. We changed it in the following manner: in order for the programmer specify the consistency required for each object, it simply adds the consistency parameter when calling the constructor of a Parse object, at the time of its creation. Then, on each request to Parse, since it is using the REST API, it adds an extra header to the request, containing the consistency parameter previously added upon the creation of that Parse object.
5 Evaluation

Contents

5.1 Setup ................................................................. 35
5.2 Latency ............................................................... 36
5.3 Throughput ......................................................... 38
5.4 Analyzing the results .............................................. 40
In this chapter we present the evaluation of our proposed solution. In the following experiments, we introduce the concept of "inconsistency". Our solution interprets the concept of "inconsistency" as when, on a GET or PUT request, the hash retrieved from ZooKeeper, which represents the most recent version of its correspondent object, is not equal to the hash stored on the correspondent object, retrieved from MongoDB. This chapter is organized in the following manner: On section 5.1, we present the setup of the experiments environment; On section 5.2, we present the results of several measures on the latency of client requests. On section 5.3, we present the results of several measures on the throughput of Parse Server, meaning how many operations it can handle per second. Finally, on section 5.4 we analyze the results from the experiments and we took conclusions out of it.

5.1 Setup

The experiments that we present next were executed locally, meaning on one computer with an Intel(R) Core(TM) i7-7500U CPU @ 3.50GHz, 16GB DDR4 1GHz of RAM, a Samsung SSD 840 EVO 250GB and with Windows 10 Home. Regarding all the components of the Parse Server + ZooKeeper system, presented in section 3.2, they are deployed in the following manner and quantity: One MongoDB replica set with three replicas, one primary and two secondaries, was deployed on Windows 10 Home; On a virtual machine inside Windows 10 Home, with Ubuntu 16.04.3 LTS, 6GB of RAM and using the maximum CPU power allowed by VMware, we deployed one Parse Server instance and one ZooKeeper server; Seven clients, which are terminals that send requests directly to Parse Server, each using its REST API, were independent virtual machines inside windows 10 Home, each one with Lubuntu 16.10, 512MB of RAM and minimum CPU power allowed by VMware.

Regarding the code changed on Parse Server, there are two parameters that need to be configured in order for our proposed solution to work. One of them is the number of old versions of Parse objects. For this setup, the minimum number of old versions is 80. This high value of old versions is justified by the fact that the communication between all the components of the Parse Server + ZooKeeper system is local and the specs of the computer that we used are high, meaning that the hardware is fast, and so that value was the minimum that works in order to resolve all consistency issues and do not get a timeout, which would cause the request to fail. For these experiments, 80 versions of the same object will be stored on MongoDB, and the size of each object will be up to 800KB larger, since it must store all the hashes (ids) of the 80 versions. The other parameter is the timeout for the operation of retrying the search for an object, which we set to 5 seconds.
5.2 Latency

We executed two experiments to measure the overhead in terms of the latency of our proposed solution. By latency we mean the interval, measured in milliseconds, between sending a request to Parse Server and receiving the response. The first experiment compares the latency of each request when the programmer sets the consistency to Default, provided by Parse Server, versus when he/she sets it to Sequential, provided by ZooKeeper. The second experiment compares the latency between requests made with: Default consistency and without occurring inconsistencies, meaning that two different objects were modified and read; Sequential consistency and without occurring inconsistencies; and Sequential consistency and with inconsistencies occurring, meaning that the same object is being modified and read by two clients, concurrently.

5.2.1 Default vs Sequential

This experiment compares the latency of the requests that are sent requiring Default consistency, which means that Parse Server will behave as it was behaving before, calling only MongoDB, versus Sequential consistency, where there are made some additional calls to ZooKeeper and MongoDB. We executed these experiments in the following manner: One client was used to send 500 requests of the same type (POST, PUT or GET) to Parse Server, one at a time, then we measured the latency of each request on the client side, and then we calculate the average from those latencies. That was done six times, two per request, alternating the consistency. Since, in order to measure the latency average, each request needs to wait for the previous to finish, it is not possible for inconsistencies to happen, because there are no read and writes at the same time.

The results of this experiment are presented on Figure 5.1, which show that exists a significant increase of around 50% more latency on a PUT request when using Sequential consistency, compared to using Default consistency. This happens because, on the method from Parse Server corresponding to the PUT request (update), four additional calls are made, two to MongoDB and two to ZooKeeper. The other requests show a less significant increase on latency, about 18% more latency on a POST request and about 10% more latency on a GET request.

5.2.2 Impact of inconsistencies

For this experiment, two clients were used and the goal was to compare the latency between requests: requiring Default consistency and without occurring inconsistencies; requiring Sequential consistency and without occurring inconsistencies; and requiring Sequential consistency, with inconsistencies occurring. We executed this experiments in the following manner: For the GET request, one client sent 500 PUT requests and the other sent 500 GET requests, concurrently. For the PUT method, both
clients sent 500 PUT requests each, concurrently. We executed this three times for each type of request (GET and PUT), alternating the consistency and the target objects. To force inconsistencies to occur, both clients must send requests targeting the same object.

The results of this experiment are presented in Figure 5.2. About the results on the latency of the PUT request, there exists a significant increase of around 74% more latency when using Default consistency and without occurring inconsistencies, compared to using Sequential consistency and without occurring inconsistencies. This value was expected to be high since the previous experiment from section 5.2.1 showed also a significant increase on the PUT request latency. There exists a significant increase of around 114% more latency when using the Default consistency and without occurring inconsistencies, compared to using Sequential consistency and with inconsistencies occurring. That is explained by the fact that, when an inconsistency occurs, Parse Server repeatedly calls MongoDB until it finds the object, as explained on section 4.1.3. Since only 19 inconsistencies occurred in 500 requests, that means that, on average, each inconsistency increases around 2% the latency of each PUT request. About the results on the latency of the GET request, there exists a less significant increase of around 30% more latency when using Default consistency and without occurring inconsistencies, compared to using Sequential consistency and without occurring inconsistencies. There exists a significant increase of around 98% more latency when using Default consistency and without occurring inconsistencies, compared to using Sequential consistency and with inconsistencies occurring. The justification for that significant increase is the same presented for the PUT request, since it first calls the method associated with the GET request, as explained on section 4.1.4. Since only 9 inconsistencies occurred in 500 requests, that
means that, on average, each inconsistency increases around 8% the latency of each GET request.

![Figure 5.2: Default vs Sequential w/o inconsistencies vs Sequential w/ inconsistencies](image)

### 5.3 Throughput

We executed two experiments to measure the throughput of our proposed solution. By throughput we mean the number of requests that Parse Server can respond, per second. Both experiments compare the throughput of Parse Server when multiple clients are sending GET or PUT requests, depending on the experiment, and requiring the Default consistency versus requiring Sequential consistency. Since, for this experiment, we wanted to avoid inconsistencies to occur, the requests were sent to different objects. These experiments were executed using a number of clients that was increasing from one to seven, all sending requests concurrently. We measured seven time the throughput with the consistency set to Default and another seven with the consistency set to Sequential, increasing the number of clients. Each client sent 500 GET requests at a time, for each measurement. The exact same setup was used for the experiments using PUT requests.

Regarding the results when sending only GET requests, as Figure 5.3 shows, it does not exist a significant difference on throughput, between sending the requests with Default consistency versus with Sequential consistency. That is explained by the fact that, on the method from Parse Server corresponding to the GET request (find), when there are no occurrences of inconsistencies, only one additional call is made, and that is to get the hash from ZooKeeper.
Regarding the results when sending only PUT requests, as Figure 5.4 shows, there exists a very significant difference between sending the requests requiring Default consistency, where the peak throughput is around 150 ops/s, versus requiring Sequential consistency, where the peak throughput is around 85 ops/s, which means that there is a decrease on throughput of about 76%. That decrease is explained by the fact that, on the method on Parse Server corresponding to the PUT request (update), when there are no occurrences of inconsistencies, four additional calls are made, two to MongoDB and two to ZooKeeper.

**Figure 5.3:** Throughput when sending GET requests

**Figure 5.4:** Throughput when sending PUT requests
5.4 Analyzing the results

We evaluated our proposed solution in order to measure how it impacts the latency of Parse requests and on the throughput of Parse Server. About the results without inconsistencies occurring, overall they show an impact on latency and throughput that is not significant, when requiring Default vs Sequential consistency. However, the results on the PUT request experiments show a significant impact on the average latency and throughput. Specifically, there exists a decrease of around 76% on the average throughput and an increase between 50 to 74 percent on the average latency. Regarding the results when inconsistencies occur, at a first glance, they seem to show a very significant impact on latency. When requiring Default vs Sequential consistency, there exists an increase of around 98% on the average latency of GET requests, and an increase of around 114% on the average latency of PUT requests. However, when carefully analyzing these results, one needs to take into consideration that there is no knowledge about what is the percentage of inconsistencies in the total amount of requests to Parse, because that number depends on many variables, and so we can not truly know what is the real impact of inconsistencies on the overall latency and throughput of Parse Server since it depends on how each Parse application is using it. Thus, we came to the conclusion that the impact of inconsistencies can only be calculated per application, by measuring the latency and throughput on real world deployments, with real users, or something that simulates that real world behavior.
6

Conclusion

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Contributions</td>
<td>43</td>
</tr>
<tr>
<td>6.2 Future Work</td>
<td>44</td>
</tr>
</tbody>
</table>
In this final chapter, we conclude this thesis with a short summary, highlighting our contributions. This chapter ends with what we propose to be the future work to the solution this thesis proposes.

6.1 Contributions

We investigate on the consistency guarantees of Parse Server and we came to the conclusion that they are dependent on how Parse Server is setting some parameters that have impact on the consistency of its storage system, MongoDB, but we could not find any information about that configuration. We also found that tuning these parameters might not guarantee that consistency anomalies like lost updates, dirty reads or stale reads are averted.

In order to improve the consistency guarantees of Parse Server, we analyzed several potential solutions to provide different consistency models on its objects, by understanding their advantages and limitations. After discuss their the pros and cons, we decided to implemented a solution which allows the developers to choose the consistency guarantees they require, per object, and offer another consistency model, provided by ZooKeeper. ZooKeeper works as a synchronization layer between Parse Server and MongoDB, by being considered the "Source of truth" regarding object versions. This solution enables Parse Server to provide Sequential consistency on its objects, if developers require it.

We evaluated our proposed solution in order to measure how it impacts on the latency of Parse requests and on the throughput of Parse Server. About the results without inconsistencies occurring, overall they show an impact on latency and throughput that is not significant, when using the original Parse Server compared to using ZooKeeper, to assure sequential consistency of Parse objects. However, updates to Parse objects have a significant impact on the performance of Parse Server, when Sequential consistency is required. Regarding the results when inconsistencies occur, we came to the conclusion that the true impact can only be known if one measures the latency and throughput on real world deployments, since it depends on the percentage of inconsistencies from the total amount of requests that are sent to Parse Server. Despite that significant impact on latency and throughput of Parse Server when update requests are sent requiring Sequential consistency, due to the well-known trade-off between consistency and performance, that is the price that the developers have to pay in order to achieve sequential consistency on the data of their apps.

In conclusion, we think that our proposed solution is a viable extension to improve the data consistency of Parse Server since: it offers an alternative consistency model to the developers, besides the one that Parse Server already offers; That consistency model is provided by ZooKeeper, which yield strong guarantees on the consistency of its data; The modifications to Parse and its SDKs are minor and local; and the impact on the latency and throughput of Parse Server is the price that developers have to pay in order to achieve a higher level of data consistency on their apps.
6.2 Future Work

In this final section, we present what needs to be done next on our proposed solution, from our point of view, in order for our extension have greater probability of being accepted by the Parse Server community.

In order to evaluate our solution, we changed the methods `createObject`, `findOneAndUpdate` and `find`, on the file "MongoStorageAdapter.js" from the Parse Server open source project. These methods, and some others related to them, need to be refactored according to the pseudocode presented on section 4.1 because, right now, the code is difficult to read and maintain. Also, in order to Parse Server work properly when consistency is specified, two more methods from the same file must be changed: `deleteObjectsByQuery` and `updateObjectsByQuery`; That is due to the translation to REST requests of all of these methods be the following: `createObject` = POST; `findOneAndUpdate`, `updateObjectsByQuery` = PUT; `find` = GET; `deleteObjectsByQuery` = DELETE.

After refactoring those methods, there are still work to do, since our proposed solution is lacking on error/exception handling, that is needed when the calls to MongoDB and ZooKeeper fail for some reason. That handling is crucial to achieve Sequential consistency on Parse objects. While writing the code for handling error/exception, our proposed solution must be further tested in the meantime. Possible tests could be: find a consistency case where there are some inconsistencies using the Default consistency of Parse Server, and guarantee that, using ZooKeeper, those inconsistencies do not occur anymore; Determine how many ZooKeeper servers are needed when an instance of Parse Server is responding to requests at its maximum load, meaning reaching its maximum throughput; Measure how the latency increases when the size of the object increases as well.

After making sure that the system Parse Server + ZooKeeper is working properly, all the others Parse SDKs must be changed in the same way the Parse Android SDK was changed, in order to our solution be available for all the applications that use Parse Server.

There is at least one performance optimization that can be done on our proposed solution. An implementation of a cache on Parse Server can improve the performance of the GET, PUT and DELETE requests, when Sequential consistency is required. The idea is the following: A cache containing the most recent objects could be implemented. Then, on each request, instead of making the initial call to MongoDB to get the object, Parse Server could first call ZooKeeper and compare if the version returned by it is equal to the version of that object that is in cache. If it is, then there is no need to call MongoDB.

Finally, more consistency models can be implemented on Parse, in order to increase its flexibility regarding consistency. One idea could be set the consistency parameters of MongoDB, as this article recommends [14], and test what are the consistencies grantees it provides. There is a probability that, depending on how those parameters are set, it could provide different consistency models, so that possibility can also be tested.
Bibliography


