NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

NUNO MIGUEL FIÚZA MARTINS BOAVIDA DOS SANTOS

Bachelor in Computer Science

DATA MANAGEMENT FOR LOCATION-DEPENDENT MOBILE APPLICATIONS

THESIS REPORT

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
March, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

DATA MANAGEMENT FOR LOCATION-DEPENDENT MOBILE APPLICATIONS

THESIS REPORT

NUNO MIGUEL FIÚZA MARTINS BOAVIDA DOS SANTOS

Bachelor in Computer Science

Adviser: Nuno Manuel Ribeiro Preguiça
Professor Catedrático, FCT-NOVA

Co-adviser: João Carlos Antunes Leitão
Professor Associado, FCT-NOVA

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
March, 2023

Data Management for Location-Dependent Mobile Applications

Acknowledgements

I would like to thank Luís Silva for all his help during the development work of this dissertation. Without his support, and that of my advisers Nuno Preguiça and João Leitão, this thesis work would not have been possible. Furthermore, this work is partially supported by NOVA LINCS (UIDB/04516/2020) with the financial support of FCT.IP.

Abstract

Over the last decade, the number of mobile applications that rely on geolocated data has increased substantially. In many cases, the users of such applications demonstrate a greater interest in data that is located in their close vicinity, than that which is located further away from them, whether it be in terms of detail or in terms of the freshness of the data.
In this dissertation, we present FocusDB, a data management system designed to support this type of application, as well as the interaction between clients and servers based on geolocated data. This system proposes a novel data model where the information is kept with several levels of detail, with the applications defining the relevant levels. The clients access the data in such a way that the distance of the client to the data object can dictate its level of detail. Additionally, the users are notified whenever their data is modified, avoiding them having to repeatedly request the server for updates.
The performance of the system shows gains in performance when compared to industry standards and opens the door for more research endeavors related to location-aware distributed systems.

Keywords: Distributed systems, dynamic consistency, geographic data, replication, mobile applications

Resumo

Na última década, o número de aplicações móveis que utiliza dados geo-localizados tem aumentado substancialmente. Em muitas destas aplicações, os utilizadores demonstram um maior interesse nos dados situados na sua vizinhança do que naqueles que se encontram a uma maior distância de si, quer em termos de atualização da informação, quer no detalhe dos mesmos.

Nesta dissertação apresentamos o FocusDB, um sistema de gestão de dados desenhado para suportar estas aplicações e a interação de clientes e servidores com dados geo-localizados. Este sistema propõe um modelo de dados onde a informação é mantida com diferentes níveis de detalhe, com as aplicações a definirem os níveis relevantes. Os clientes acedem à informação geo-localizada, obtendo diferentes níveis de detalhe dependendo da distância a que os diferentes objetos se encontram. Adicionalmente, os utilizadores são notificados sempre que os seus dados são modificados, evitando que estes tenham de repetir os pedidos ao servidor.

O desempenho do sistema mostra ganhos ao nível do desempenho quando comparado aos padrões utilizados na indústria e abre a porta a mais estudos futuros acerca de aplicações distribuídas com noção da localização dos dados.

Palavras-chave: Sistemas distribuídos, coerência dinâmica, dados geo-localizados, replicação, aplicações móveis

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

1.1 Context

Over the past decade, cloud-based platforms have risen to the forefront of web application development, due to the many benefits they grant over locally hosted applications. Providers such as Amazon Web Services, Microsoft Azure and Google Cloud offer computing as a utility, which can be scaled and payed for according to the needs of the developer, even if those needs change over time, which, in turn, helps lower the infrastructure costs for the clients. Cloud services are also highly reliable due to the redundancy built into their systems, where data is replicated in multiple data centres in the same region and/or different regions. This not only allows for high levels of service up-time and availability even in the presence of failures, but also for low latency access to the data because of its increased proximity to the end-user.

Cloud platforms have also recently even given another step forward by starting to employ *edge servers*. These are computing and storage nodes with less resources than centralized cloud data centers but are more numerous and located even closer to the end-users, which allows platforms to be able to provide even lower latency to their customers.

On the other hand, modern distributed applications have been increasingly shifting towards supporting greater interaction between users. This tendency is mainly powered by the ubiquitous and highly capable mobile devices which constitute the primary target for engagement with such novel applications. It is now more imperative than ever, that the access to shared data among mobile devices be as efficient as possible, in order to allow a more comprehensive scalability of these systems.

Real-world examples of such applications include multiplayer games such as *PokemonGo* [32], maps services with real-time crowd-sourced traffic information and alerts such as *Parkify* [34], *Lime* [23] or *Waze* [57], or any data distribution system serving localised content such as ads, local news alerts, or social media features.

1.2 Motivation

One factor that is common among all these applications is that data objects are tied to specific geographical locations. Because of this property, the location of the users becomes an important factor regarding the data objects that are accessed and modified by those users. The heuristic established suggests that users are far more likely to be interested in data that is closer in proximity to their location, than in data that is more distant.

This offers the opportunity to design distributed data storage systems that can be optimised by taking this property into consideration, when dealing with replica placement and access pattern prediction. Maybe less obviously, this also offers the opportunity to have specialised data replication protocols that take into consideration the location associated with data objects, since users will be less concerned about the detail or freshness of data that is far away from their current positions (or to their destination, when moving). This allows to focus the operation of a distributed data management system, and associated replication protocols, in providing the greater detail and improved data freshness for data that is associated to locations in the vicinity of users, which will be of primary interest to them.

When analysing the currently available commercial solutions, such as Cassandra [19], MongoDB [47] or CosmosDB [48], it is clear that they are designed for a more generic use and, as such, are not optimized to operate in the mobile scenarios described previously, failing to take into account the locality of the data as well as the users' in their data models and replication protocols.

Similarly, current consistency models from strong consistency to weak consistency come up short when met with the requirements imposed by the mobile user data access patterns, since they also do not consider data properties such as location to define the guarantees that are provided to users that access replicas. Consequently, they cannot adapt their behaviours or provide clear guarantees regarding the location associated with data objects and the location to where data is replicated being it other servers or mobile clients. [49] This hinders the opportunity to further improve the performance and scalability of the current state of the art solutions at best (in terms of latency, throughput, and overall usefulness of data exposed to clients), and can lead to performance degradation and lower user experience at its worse.

1.3 Contributions

To respond to this need, this dissertation presents FocusDB, a first step in a wider research effort dedicated to propose novel data consistency models and replication protocols that use location associated with data, server replicas, and users as a primary factor to govern the data management strategy. FocusDB achieves this first step through two main contributions. The first is a novel data model that is: *i*) capable of adapting the consistency guarantees enforced on data exposed to clients, by taking into account the location

associated to data and the indications of the user related to locations of interest; and *ii*) automatically adjusts the detail of exposed data, through aggregations, that are guided by the location associated to the data and the user. The second contribution is a client data caching mechanism that leverages the locations of interest provided by the user, to ensure good performance even in low connectivity scenarios.

The initial prototype of FocusDB was developed using MongoDB as the underlying technology for the system storage layer. However, since the data model is agnostic to the underlying storage layer, several other database types, such as geospatial databases, could have been explored.

1.4 Publications

The contents of this dissertation were exhibited in two scientific papers. The first was submitted to the Inforum 2022 Computer Science Symposium. It was accepted and presented to the attendance of the Inforum 2022 event, which took place on the 8 and 9 September 2022, in Politécnico da Guarda, having been later published in *INForum 2022, Atas do 13º Simpósio de Informática (2022)* [43]. The second paper was submitted to PaPoC 2023, the 10th Workshop on Principles and Practice of Consistency for Distributed Data, and it was also accepted by the program committee, having also been presented on May 8th, 2023, in Rome [44].

1.5 Document organization

The remainder of this document is organized as follows:

- **Chapter 2** details the related work - some of the key concepts to this dissertation and implementations of other authors that can help understand common problems in this work;
- **Chapter 3** describes the design of the system as a whole, its goals and purposes, as well as all of its sub-components;
- **Chapter 4** presents a deeper understanding of the system components by delving into some of the implementation details;
- **Chapter 5** evaluates the performance of the system as a whole when compared to similar current implementation strategies;
- **Chapter 6** concludes the dissertation based on the development of the work presented, as well as the results obtained in the evaluation stage and describes some of the projected future work;

2

Related Work

In this chapter, we present in greater detail some of the most relevant concepts to this thesis work.

In section 2.1 we take a closer look into distributed system architectures, starting with centralized solutions and then branching further onto decentralised and hybrid ones.

Next, in section 2.2 we examine the underlying technologies that power the operation of cloud systems, while in section 2.3 we explore some of the methods that have allowed the development of mobile applications.

In section 2.4 we compare different consistency models along the consistency spectrum, ranging from strong to weak, as well as some hybrid models.

In section 2.5 we study some specific implementations that may be of interest to this dissertation and that relate to several of the topics covered previously in this chapter.

Finally, in section 2.6 we explore the details of some database options for the future implementation of the thesis work.

2.1 Distributed System Architectures and Models

A computer network architecture pertains to the organization of a distributed system, which consists in the definition of its components and the ways in which they interact with each other, as well as their roles and locations. The characteristics of an architecture dictate the trade-offs between its properties, such as performance, reliability and security [11].

2.1.1 Centralized

2.1.1.1 Client-Server

The most simple architecture is the client-server model because it separates the concerns of a client, the end-user of data, and those of a server, the data source. It consists of having many client nodes performing requests to a single central server node, where the server executes the operations requested and returns a response with the appropriate result.

Since data is stored only in a single node, it allows for easier data back-ups as well as better security to its resources. Furthermore, a server dedicated to performing requests can improve the overall performance of the system. However, the centralisation of data in a single node means that there is only one point of failure, which can render the entire network inoperable. It also fails to scale beyond a certain threshold, since the resources of a single server can become limited and cause a bottleneck.

2.1.1.2 Client-Server partitioned/replicated

To address these shortcomings it is possible to run several server nodes, to better distribute the load between machines. These servers can be partitioned, replicated or both. Partitioning means that nodes only store a portion of the total data and, consequently, can answer only to requests for that data. However, not only does it not guarantee a distribution of requests since a large amount can pertain to the same partition, it also causes some of the data to become unavailable if a node fails.

Another solution is to replicate the data in all servers, meaning all requests can be processed by any given node. It provides redundancy, which expands the number of failure points and allows for load distribution, with the cost of having to coordinate the data state in all replicas.

In practice, partitioning and replication are usually combined by replicating each data partition to several nodes. The data can also be geo-replicated, meaning that the replicas are placed strategically in several geographical locations, in order to reduce the distance to clients in a given region.

2.1.1.3 Client-Server with caching

The most simple version of a client, called a *thin client*, consists of an interface that only performs requests to the server. This can be detrimental since the client places a large load on the server which can have an impact on latency and interactivity. To relieve some of the server work, it is possible to perform caching, either in the client or in proxy servers, which saves part of the data in volatile memory which can be accessed without a request to the central server. A client-side caching approach also allows for access to the data while disconnected from the central server.

2.1.2 Peer-to-peer

In opposition to the client-server approach where the consumption and supply of resources is divided, a peer-to-peer architecture consists in a system where all nodes act both as clients sending requests and as servers responding to those requests. These nodes are called peers because they possess equal privileges and responsibilities for processing the data in the network [41]. Tasks are highly decentralised, meaning that the computation, storage and bandwidth requirements are distributed among the participants of a network.

When a node is introduced in a network, the configuration required is usually very little or nonexistent due to the network being able to self-organize. In addition, nodes are generally owned by independent individuals that join the system voluntarily instead of being controlled by a single organization. Because of this, the barrier to deployment is much smaller since there is not much investment needed for dedicated infrastructure, unlike with client-server.

It possesses a greater potential for scaling up due to the distribution of tasks, since every node contributes to the pool of resources available. Furthermore, peer-to-peer systems also tend to be resilient to failures because, on one hand, there are no or very few critical nodes, and on the other hand, in the event of an attack, to effectively have an impact on the system, an attacker must target a large proportion of the node pool.

Peer-to-peer systems are often classified along two criteria:

Degree of centralization: Classifies the presence of centralised components in the architecture.

- **Partially centralised:** There is a dedicated component with the job of maintaining the pool of nodes and controlling the system. This makes the infrastructure easier to construct but introduces some of the client-server disadvantages, such as poorer scalability and resilience to attacks since the controller acts as a bottleneck and as a single point of failure.
- **Decentralised:** There is no dedicated controller node, which removes the bottleneck and single point of failure. However, the network has to rely on flooding protocols to propagate changes to nodes, which is less efficient than having a node with direct access to the required peers. In some systems, more powerful nodes can be designated as *supernodes*, meaning that they can provide additional benefits such as storing state, keeping an index of available content and acting as a *rendez-vous* point for nodes behind firewalls. These nodes can improve system performance but can also increase vulnerability to node failure.

Network topology: Classifies the organization of the network in terms of the connection of its peers.

- **Unstructured:** Links between nodes are formed in a non-deterministic manner, meaning that the overall graph does not have any particular structure. When a new member joins the network, it only obtains the information of a small set of neighbouring nodes, which gives it only a limited view over the system. Because of this, the transmission of requests is less efficient since it has to be conducted through flooding. Nonetheless, it better supports the entry and exit of nodes in the system, called *churn*, which increases the network robustness to very dynamic environments.

- **Structured:** Each node has a unique identifier chosen uniformly from a large numeric key space that determines its position within the network as well as constraining its set of links. A data structure, such as a distributed hash table, is used to perform efficient key-based routing, which, given a starting node and a key, produces a set of steps going from peer to peer that ends in the node responsible for that key. This method of routing is much more efficient than flooding. However, when in conditions where churn is high, the performance is impacted due to the data structure having to be updated every time a node enters or leaves the network.

2.1.3 Hybrid architectures

There are also some networks that choose to combine features from client-server architectures and others from peer-to-peer ones, resulting in hybrid solutions [51].

One approach may be to have a peer-to-peer network work as a service and attend to requests from clients from outside the network. This, however, limits the number of nodes that constitute the peer-to-peer network.

Another approach is to separate some functionalities between the two architecture types, combining the advantages of each one. Client-server architectures are particularly effective in dealing with searching and indexing because the centralised nature of the system allows them to answer these types of queries in constant time. Peer-to-peer solutions on the other hand, can withstand more demanding scenarios, such as when there is a high demand for network resources and bandwidth, since the cooperation of multiple nodes allows the allocation of resources from each peer to achieve better scalability. One example of this hybrid architecture is called Legion, and is further detailed below in section 2.5.4

Finally, the *supernodes* referenced in section 2.1.2 can also be considered a hybrid solution since they take the role of centralised entities, being responsible for storing state and indexing, among other tasks.

2.2 Infrastructure

Another important subject to explore when developing a distributed application is the the underlying infrastructure over which the system operates. Over the years, the technology has evolved to form a hierarchy where the cloud is situated at the center and less powerful nodes are distributed in the periphery to improve latency. In this section, we will be detailing some of the components of the overarching cloud infrastructure.

2.2.1 Cloud

The cloud is the central part of the infrastructure and it consists of multiple datacenters situated in strategic positions, in order to better provide its services to its surrounding geographic area, the *availability zone*. Availability zones are physically separate locations,

consisting in one or more datacenters equipped with independent power, cooling, and networking. Several availability zones compose a *Region*, which connects the various availability zones through a dedicated regional low-latency network [2].

Datacenters are extremely powerful in terms of computing, storage and networking capabilities in order to handle vast data loads and act as the core of the network. They are capable of replicating their data on different sites, not only to provide low latency access to data, but also to guarantee high availability even in the presence of faults.

2.2.2 Edge Servers

Having a network of cloud servers distributed in key geographic locations is essential for providing low latency services across regions or even the globe. However, inside of a region, locations further away from the central server are still subject to higher latencies than less distant sites. It would be advantageous to have server instances more distributed throughout a region but, since cloud server data centers usually possess large amounts of computational resources and are in general, very expensive to construct, it is a tough endeavor to do so in such a scale.

Therefore, smaller and more easily manageable server nodes are a fitting compromise, since they are more easily distributed, whether through compact data centers or through stand-alone servers, while also being capable of storing data and responding to client requests with a low latency. These are usually called edge servers, due to being located further away from the center of the cloud, the so called *edge* of the network.

Edge computing [45] is an important aspect of this architecture style, specifically for IoT, and consists of performing computations over data closer to the origin point of such data. This can prevent sending tremendous quantities of data to the cloud, processing the data or answering the requests immediately at the edge and, in turn, saving bandwidth and processing time in the central nodes. The cloud is, however, still used for a subset of requests as well as storage needs, since at the edge, although scalability is more easily achieved, the amount of resources is still more limited than in the cloud servers.

2.2.3 Content Distribution Networks

Content Distribution Networks (CDNs) are a particular use case of the cloud-edge hierarchical architecture, where the edge servers collaborate to provide fast delivery of static data. The CDN allows for the quick transfer of assets needed for loading Internet content such as HTML pages, javascript files, stylesheets, images, and videos. When a server receives a request for some data, it first checks any of the CDN server has that particular data cached and only if the result is negative, does it forward the request to the central servers.

This results in lower latency to the end-users, especially when multiple round-trips are required to load content, large scaling to better handle instantaneous high loads and reduced traffic to the central servers.

2.3 Mobile Applications

Mobile applications are computer programs designed to run on mobile devices, like smartphones and tablets. They differ from standard desktop applications in the sense that the latter requires the user to be fixed in the vicinity of the machine in order to interact with it, while the former gives the user more freedom in terms of the environments in which they can have their interactions. This separation in use cases impacts not only the human-machine interfaces but also allows the applications to explore different scenarios, namely with location-based services like GPS.

Besides the development of the cloud-based technologies, which have undoubtedly helped foster the expansion of the mobile application market, other technologies have also had an impact in this endeavor. Below are some examples of such developments.

2.3.1 Caching

One of the motivating factors behind caching is a paradigm named *offline-first*, which dictates that because network coverage can vary abruptly and unpredictably in a remote environment, applications should be designed to function without connectivity. To this end, applications can cache data, that is, save information in volatile memory after the user has accessed it or even before the user needs it, in preparation for moments of disconnection. Not only can this approach mask poor connectivity, it can also be an effective tool for reducing latency, since a data item can already be present in the device's cache, invalidating the need for a remote request.

Despite this, it also possesses some drawbacks since, if a data item is updated and it also present in the cache, that entry must either be invalidated or updated. Furthermore, if this update occurs in a period without connectivity, it may lead to a conflict of updates, which requires the employment of a conflict resolution policy, such as *first-write-wins* or *last-write-wins*.

2.3.2 Geofencing

Geofencing is a technique that allows for the definition of virtual perimeters in real-world geographic locations. It employs technologies like the global positioning system (GPS), radio frequency identification (RFID) or IP addresses to define the edges of the perimeters, which can be used to track the physical location of a device in the particular region of the fenced area.

These virtual barriers can either be active, where the end-user has to opt-in to location services and have the corresponding mobile application open or passive, where the tracking is always on in the background and relies on Wi-Fi and cellular data instead of GPS or RFID.

Geofences can also be set up so that when a device enters or exits the perimeter, it raises an alert, which allows for applications to react to the movement of the device.

2.4 Replication

In a distributed system, it is natural for there to exist multiple copies of the same data in different machines. This is because having a single machine being in charge of the storage of data would not only cause a tremendous bottleneck in performance, it would also mean that in the event of a failure, the entirety of the data would become unavailable. Therefore, the best course of action when in pursuit of availability is to replicate the data on several nodes. This, however, raises another issue: if there are multiple instances of data, how does one guarantee those instances remain equal, i.e. consistent, across all replicas? There are several approaches, which we will detail further in this section.

The CAP theorem [12] states that any distributed data store can only provide two of the following guarantees:

- Consistency - All clients see the most recent version of data possible.
- Availability - Every request sent to an online server will be answered.
- Partition Tolerance - The system is kept functional in the presence of network partitions.

In reality, partitioning is to be expected in large scaled systems, mainly due to high latency or node unavailability. This means that these systems must be prepared to handle these situations, and therefore the choice presented the CAP theorem is reduced to either consistency or availability.

Classic ACID-based database systems opt for consistency instead of availability. However, more recently with the emergence of highly responsive web-applications, the focus has shifted towards a more availability-centric focus in order to keep interactibility high.

2.4.1 Strong consistency

In some systems, it is imperative that data stays as consistent and up-to-date as possible. To achieve this, consistency is put before availability by imposing that every operation be read by all users in the same order, so that the state remains consistent between replicas and giving the illusion that there is only a single node. These are some of the models capable of achieving strong consistency:

2.4.1.1 Sequential Consistency

Sequential consistency imposes that operations in the system are performed in a total order, which is consistent with the order that is observed in every single replica. It is not restricted by a real-time order, only that the operations respect the total order [20].

As an example, if two replicas were to execute one operation each, *e1* and *e2*, both orders *[e1, e2]* and *[e2, e1]* would be sequentially consistent, as long as both replicas respected that order individually.

2.4.1.2 Linearizability

Linearizability consists of two guarantees: a user always reads the most recently committed version of an item and never sees an uncommitted or partial write [16]. It is stronger than sequential consistency because, unlike it, it is restricted by a real-time order, as well as a total order.

Continuing the previous example, if the real-time order of the operations were *[e2, e1]*, that is, if the result of *e2* has returned before a request for *e1* is issued, then the only valid linearizable order would be *[e2, e1]*, since that is the real-time ordering in which the operations were issued.

2.4.1.3 Serializability

Serializability differs from the two previous models in the sense that it is a transactional model, and therefore has a different granularity. This means that it concerns multiple operations over multiple objects, unlike the former single-operation, single-object. It guarantees atomicity of each transaction and that the execution of a set of transactions over multiple items is equivalent to some serial execution, i.e. total ordering, of the transactions.

It is, however, not constrained by a real-time order. It is possible to impose a real-time order in *Strict Serializability*, a combination of serializability and linearizability.

2.4.2 Weak consistency

In contrast to the previous consistency level, weak consistency places availability over consistency, where it is preferred to act on operations as soon as they are requested. This implies that different replicas can reflect different states at any given moment, since there is no guarantee of a single ground truth. Furthermore, no order of operations is respected and it is possible to receive outdated reads.

2.4.2.1 Eventual Consistency

Eventual consistency is based on the concept that availability is key and that the state of a replica is permitted to diverge from that of other replicas'. This inconsistent state allows for reads to be received with outdated values or with different orders. However, if there are no updates being issued, the system will eventually converge to a common state among all replicas [56].

To ensure this convergence of replica states, a conflict resolution mechanism needs to be employed, which consists of two phases. The first is known as *anti-entropy* [7] and consists of an exchange of the versions of replica data and the second, *reconciliation*, deals with selecting the final state when concurrent updates have occurred, according to some application-specific criteria. One common method is using a policy *last-writer-wins*, as well as more recently CRDTs, as outlined in section 2.4.2.3.

2.4.2.2 Casual Consistency

On the other hand, casual consistency is more strict than eventual consistency. It states that operations that are casually-related to one another should appear in the same order on all replicas. However, operations that are causally unrelated, i.e. concurrent operations, can appear in different orders in different replicas.

This requires for there to be a mechanism able to track the casual dependencies between operations, so that the order of those operations can be agreed upon universally in such a way that said dependencies are respected. This mechanism is called the *happens-before* relation [21] and consists of the following:

- $a \rightarrow b$, if a and b are operations performed on the same replica and a preceded the occurrence of b;
- $a \rightarrow b$, if a and b are operations performed on different replicas and b is the reception of the message sent by event a.

2.4.2.3 Conflict-free Replicated Data Types

As mentioned previously, systems that adopt a weak consistency model allow replicas to temporarily diverge, requiring a mechanism for merging concurrent updates into a common state. Conflict-free Replicated Data Types (CRDTs) [36] provide a principled approach to address this problem as abstract data types, with well defined interfaces, designed to be replicated at multiple nodes and exhibiting the following properties:

- Any replica can be modified without coordinating with any other replicas;
- When any two replicas have received the same set of updates, they reach the same state, deterministically, by adopting mathematically sound rules to guarantee state convergence.

Preguiça divides the concerns of CRDTs between application, system and CRDT developers. Application developers are mostly concerned with the functionality provided by the CRDT and the data structures that can be used to maintain the state of the application, whereas the system developers are more concerned with the synchronization model, that is, how to to guarantee that the replicas of CRDTs in the multiple nodes of the system are kept synchronized.

The synchronization model has to guarantee that all updates reach all replicas and that those replicas all converge to the same state. This can be achieved through several synchronization approaches:

State-based synchronization Replicas synchronize by establishing bi-directional (or unidirectional) synchronization sessions, where both replicas send their state to a peer replica. When a replica receives the state of a peer, it merges the received state with its

local state. CRDTs designed for state-based replication have to define a merge function to integrate the state of a remote replica.

Operation-based synchronization Replicas converge by propagating updates to every other replica. When an update is received in a replica, it is applied to the local replica state. Besides requiring that every update operation is reliably delivered to all replicas, some CRDT designs require updates to be delivered according to some specific order, with causal order being the most common. CRDTs designed for operation-based replication must define, for each update, a generator and an effector function.

Delta-based synchronization If an update only modifies part of the state, propagating the complete state for synchronization to a remote replica is inefficient. On the other hand, if a large number of updates modify the same state, e.g. increments in a counter, propagating the state once is much more efficient than propagating all update operations. Delta-state CRDTs address this issue in a principled way by propagating *delta-mutators*, which encode the changes that have been made to a replica since the last communication. The first time a replica communicates with some other replica, the full state needs to be propagated.

Pure-based synchronization The operation-based CRDTs require executing a generator function against the replica state to compute an *effector* operation, which may introduce an unacceptable delay for propagating an update. Pure-operation based CRDTs address this issue by allowing the original update operations to be propagated to all replicas, typically at the cost of more complex operation representation and of having to store more metadata in the CRDT state.

This notion allows for the construction of logical clocks such as *Lamport clocks* or *vector clocks*, which, in turn can be used to record the casual history of operations.

Casual consistency is a reasonable middle-ground between strong consistency levels and eventual consistency, since it offers similar availability to eventual, while also granting additional benefits such as the ordering of casually-related events.

2.4.3 Hybrid models

As seen before with casual consistency, often the space between strong and weak consistency models can be explored in search for solutions to problems in real-world implementations. Below we expose some of the solutions that try to conjugate the two sides of the consistency spectrum and extract their advantages.

2.4.3.1 The Escrow Transactional Method

The Escrow Transactional Method [33] is designed to support long non-blocking update transactions. It ensures the recoverability of intermediate results prior to commitment so

that updates are safe against memory and media failures, as well as being able to abort transactions. If one were to lock each of the resources in the transaction, such as in a standard two-phase locking approach, in a concurrent scenario, only one transaction would be able to access the resources at a time, which would limit throughput.

Instead of issuing reads and writes, transactions can issue reads, increments, and decrements. As long as a transaction can guarantee that executing its increments or decrements will not violate its invariants no matter which other transactions commit or abort, the increments and decrements can be executed in any order, and transactions can issue their increments and decrements without locking.

An Escrow request consists of checking whether a certain quantity C can be subtracted from field F with the guarantee that condition will not be violated. If this is true, the normal processing of the request can proceed, otherwise it is aborted.

Internally, every field contains the lowest and highest values that field might assume from any combination of commits and aborts of currently live transactions with Escrow requests, as well as the value assuming all Escrow requests are committed. Furthermore, this method utilises journal entries for every active transaction on the data item to ensure recoverability.

O'Neil [33] finishes by concluding that the Escrow method provides several benefits such as not causing deadlocks, allowing high concurrency items to not act as performance bottlenecks, enabling human interaction in the midst of a transaction as well as distributed transactions.

2.4.3.2 Epsilon Serializability

Epsilon Serializability (ESR) [40] is a generalization of classic serializability, which allows some limited amount of inconsistency in transaction processing through an interface called Epsilon Transactions (ETs). Epsilon serializability enhances concurrency since some non-serializable execution schedules are permitted. For example, ETs that just perform queries may execute in spite of ongoing concurrent updates to the database, which means the query may view uncommitted data.

Managing correctness in ESR consists in bounding the amount of imported and exported inconsistency foe each ET. An update transaction may export some inconsistency when it updates a data item while query ETs are in progress up to an export limit and a query ET may import some inconsistency when it reads a data item while uncommitted updates on that data item exist up to an import limit.

Incremental accumulation of inconsistency depends on the triangle inequality property of metric spaces. Without triangle inequality the distance function for the entire history would have to be recomputed each time a change occurs.

2.4.3.3 Continuous Consistency Model for Replicated Services

Yu et al. [58] explore the semantic space between traditional strong and optimistic consistency models for replicated services, where there is a subset of applications that can tolerate relaxed consistency but also benefit from having its inconsistency bounded to a specific maximum for that application.

To this end, Yu et al. have developed TACT, which is a middleware layer that accepts specifications of application consistency requirements and mediates read/write access to an underlying data store. If an operation does not violate pre-specified consistency requirements, it proceeds locally. Otherwise, the operation blocks until TACT is able to synchronize with one or more remote replicas, as determined by system consistency requirements.

Three metrics were chosen to bound consistency: Numerical error, Order error and Staleness. Numerical error limits the total weight of writes that can be applied across all replicas before being propagated to a given replica; order error limits the number of tentative writes that can be outstanding at any one replica and staleness places a real-time bound on the delay of write propagation among replicas. Each replica contains a vector with these three dimensions which means each replica can have different consistency bounds.

Each of these metrics is bound by an algorithm, where the numerical error is bounded using a push approach based solely on local information, order error bound is enforced using a write commitment algorithm combined with compulsory write pull and staleness is maintained using a real-time vector.

In the end, the authors conclude that relative to strong consistency techniques, TACT improves the throughput of these applications by up to a factor of 10 and relative to weak consistency approaches, TACT provides strong semantic guarantees regarding the maximum inconsistency observed by individual read and write operations.

2.4.3.4 Vector-Field Consistency

Vector-Field Consistency (VFC) [42] is an optimistic consistency model that allows bounded divergence of the object replicas. It selectively and dynamically strengthens or weakens the replica consistency based on the ongoing game state in order to address two points: how the the consistency degree changes throughout and how the consistency requirements are specified. In regards to the first point, it considers observation points (pivots), around which the consistency is required to be strong and, as the distance to the pivot increases, the consistency level is weakened. Since pivots can change with time, objects' consistency needs can also change with time. In regards to the second point, It provides a 3-dimensional vector for giving a bound to the replica divergence, consisting of the delay between operations, the number of operations and the magnitude of changes.

The authors also propose an architecture implementation for Vector-Field Consistency

called Mobihoc, which consists of a middleware platform aimed at supporting the design of multiplayer distributed games for ad-hoc networks. Mobihoc enforces VFC by managing the game state between the network nodes, as well as providing programmers with the adequate means to parameterize VFC according to game semantics. It follows a client-server architecture where upon the establishment of the ad-hoc network, one of the nodes becomes the server, which then coordinates write-lock management, update propagation and VFC enforcement. The shared data is a collection of objects and each node maintains local replicas of all objects in the Object Pool container. Reads are performed locally over the replicated data but writes require a lock acquisition on the server, after which updates are propagated from the server to the clients.

2.5 Mobile Systems

In this section, we will explore some mobile system implementations. The goal is to gain further insight into solutions of similar problems present in this thesis or that attempt to solve some of the smaller challenges that compose the proposed work.

2.5.1 Bayou

Bayou [53] is a replicated storage system that was designed for an environment with portable machines where the network connectivity may be limited or intermittent. The primary objective of the system is to maximize its availability, where a client can read from and write to any given replica. This need for flexibility is best handled by a system with an eventual consistency level, where operations can be performed as soon as they are requested but where the system must guarantee that an operation is eventually reflected in all replicas. Strongly replicated systems have been shown to yield unacceptably low write availability in partitioned networks, so the goal was to keep the collaboration between replicas at a minimum, as to maximize their availability.

Bayou replicas communicate with each other with occasional, pairwise transmissions called anti-entropy sessions, to skirt the possible connectivity issues in the network, but also to minimize the pay-per-minute costs associated with some means of communication. This form of communication can originate partitions of nodes that, despite being disconnected from a majority of the network, are still able to communicate with each other, which, along with the immediate execution of operations inherited from eventual consistency, can mask periods of disconnection. Every computer eventually receives updates from every other, either directly or indirectly, through a chain of pairwise interactions.

Since Bayou only guarantees eventual consistency, applications must be ready to deal with out-of-date or out-of-order reads, as well as write conflicts. Furthermore, applications must be involved in the detection and resolution of conflicts since these naturally depend on the semantics of the application. Bayou's mechanisms allow for applications to achieve automatic conflict resolution at the granularity of individual update operations

without compromising security or eventual consistency, which is highly desirable because it enables a Bayou replica to remain available.

The automatic conflict resolution mechanism consists of two phases: dependency checks, which involves performing a query to a server and, when compared to with the modified data, checking whether it returns the expected result or not; and merge procedures, which are instructions included with every write operation that are responsible for resolving any detected conflicts and for producing a revised update to apply. When a write is issued, it is immediately performed but it is not immediately committed. The update is first deemed tentative and is then ordered according to timestamps assigned to them by their accepting servers. As the anti-entropy process continues, servers will learn of new updates performed by other servers. Write operations may need to be undone and redone multiple times as a means of organizing the update order. However, when a write becomes stable, i.e. when the set of writes that precede it in the server's write log is fixed, its state finally becomes committed.

However, in the case where automatic resolution is not possible, the merge procedure will still run to completion, but is expected to produce a revised update that logs the detected conflict in some fashion that will enable a person to resolve the conflict later. Users can still read tentative data but are aware of its situation.

2.5.2 Fidelity-Aware Replication for Mobile Devices

K. Veeraraghavan et al. [55] present a practical problem of the emerging technology of mobile devices, which is that often the resource constraints of these devices, like memory or bandwidth, force application developers to store data in formats of reduced fidelity. This means that instead of storing the entire set of data entries, like it is done on high-fidelity devices such as laptops and desktops, the low-fidelity device is obligated to summarize and restrict the number of entries in a given set of data.

A way to counteract these limitations is to replicate the data across multiple high and low fidelity devices with an eventual consistency level, so that updates can be performed even when disconnected from other nodes. The fidelity level should always be reduced when transferring from a high to a low fidelity device, as to not waste resources; updates to a set of fields must be done atomically and updates on low fidelity devices should be integrated with high fidelity data as soon as possible.

Polyjuz is a framework capable of such replication, consisting of a layer between the application and the conventional replication platform, that focuses solely on bridging devices of different fidelity levels. It does so by maintaining a separate collection of data items at each fidelity level, which is replicated by the underlying platform within that fidelity world. A high-fidelity device not only stores the collection for its native fidelity level but can also store lower-fidelity collections. When synchronizing a low level with a high level fidelity device, Polyjuz reduces the fidelity of items from the high-fidelity collection when copying and, in the reverse direction, reintegrates updated items

in the low-fidelity world with their counterparts. To separate itself from the underlying replication platforms, it keeps its own version vectors to keep track of data updates called tagged vectors, where the version number of an item is tagged with a label indicating its fidelity.

A layered design allows to keep the Polyjuz layer simple and specific to fidelity-aware operations, as well as to reuse of synchronization protocols, knowledge representation schemes and transport mechanisms, while also inheriting the benefits of the underlying replication layer such as low bandwidth consumption. It does, however, also incur some downsides. One of which consists of increased overhead storage necessities by keeping multiple representations of items on replicas that support multiple fidelity levels as well as two layers of replication metadata, while also increasing the bandwidth overhead by creating spurious conflicts when idempotent fidelity operations are performed independently on different replicas.

2.5.3 Simba

It is common for mobile applications to resort to cloud-based solutions to enable sharing of data among users and to offer a seamless experience across devices, since those are often important tenets in the context of such applications. However, the authors argue that developers continue to struggle with writing applications that are correct and consistent, since every developer tries to re-solve the same problems of data consistency.

To counteract this, Perkins et al. [35] suggest a high-level programming abstraction that provides end-to-end data consistency and is well-suited for mobile application necessities that separates itself from prior solutions in three areas: first it considers sacrificing strong consistency to be a must in this context due to the limited and intermittent nature of mobile connectivity and that weaker levels of consistency are better adapted to handle periods of disconnection; second, even if devices are connected, they don't need the strongest form of consistency possible due to bandwidth caps and interactivity goals and third, it is crucial to manage data at application-relevant granularity.

Moreover, after a study of a number of popular mobile apps, the authors came to a number of conclusions which can be grouped into two categories: inadequate data granularity and lack of end-to-end tunable consistency. To address these shortcomings, the authors developed Simba, a data cloud-based data-sync service for mobile apps based on a novel data sync abstraction called sTables.

In sTables, all user and app data is seamlessly synchronized with the cloud and other devices. sTable rows can contain structured and unstructured data, providing a data model that unifies the two, meaning that applications can store all their data in a single store. Furthermore, every sTable is associated with its own consistency level, which can be:

- Strong: all writes to a row are serializable;

- Causal: if the client has not previously read the latest causally preceding write for that row, it raises a conflict;

- Eventual: last writer wins.

Simba's architecture can be divided into two major components: a client (sClient) and a server (sCloud), which communicate via a sync protocol.

The sCloud has the responsibility of managing data between multiple sClients, sTables and Simba-apps, providing the consistency levels described previously. It consists of a data store (Simba Cloud Store) and client-facing gateways. sClients authenticate via the authenticator and are assigned a Gateway by the load balancer. All subsequent communication between that sClient and the sCloud happens through the designated gateway. The store is responsible for storage and sync of client data, for both objects and tables and sTables are partitioned across Store nodes. The Gateway manages client connectivity and table subscriptions, sends notifications to clients, and routes sync data between sClients and Store.

sClient is the client-side component of Simba which supports the CRUD-like Simba API. sClient provides reliable local-storage of data on the client, and data sync with sCloud, on behalf of all Simba-apps running on a device.

Finally, in the sync protocol, for all three consistency models, any client willing to access a table needs to register a sync intent with the server in the form of a write and/or read subscription for that particular table. On the other side of the connection, sCloud is expected to provide multi-version concurrency control to gracefully support multiple independent writers. To handle conflicts in the strong and causal consistency levels, Simba uses a versioning scheme which uses compact version numbers instead of full version vectors. The three consistency models also rely on the versioning scheme to determine the list of sRows that have changed.

2.5.4 Legion

Van der Linde et al. [25] present a peer-to-peer based framework named Legion that allows users to replicate subsets of application data locally following an eventual consistency model, which allows for asynchronous propagation of updates. This reduces the load on the centralised component and minimises update latency, as well as allowing clients to continue communicating when the connection to the server is lost.

Its system design can be divided in three areas: the Object Store, Communication and Security Mechanisms. The Object Store maintains local replicas of shared objects, where related objects are grouped together in containers. These objects are encoded in CRDTs to provide eventual convergence without resorting to strong coordination. Delta-based CRDTs are used in order to allow for efficient synchronisation, where each client keeps a list of received deltas that respects causal order.

The framework supports point-to-point and multicast communication. Each container has an associated Overlay Network over which all object updates in the container are propagated to all connected clients and which also promotes low latency links between clients. Server connections are only established by a small subset of clients, called active clients and peer connections first require an exchange of information called signaling, which is supported by the centralised infrastructure.

To ensure privacy and integrity of data from unauthorised users, each container has an associated symmetric key, generated and maintained by the centralised infrastructure. Only clients granted access to a container obtain its symmetric key, which is then used to encrypt all messages regarding that container.

Legion also uses a set of adapters to utilise existing infrastructure instead of its own standalone servers. They integrate the system with GDriveRT, providing data storage and support for legacy clients, signaling, authentication and key management and a service API.

The authors conclude that latency for update propagation is much lower using Legion when compared with the use of GDriveRT and that load to the centralised infrastructure is greatly reduced by leveraging peer-to-peer interactions, where clients are also able to interact while servers are temporarily unavailable.

2.5.5 Practical Client-side Replication for Mobile Systems

According to van der Linde et al. [24], direct client-to-client interactions (P2P) have the potential of achieving lower latency when compared with the traditional client-server approach, especially when paired with a causal consistency model. However, such a strategy poses a number of challenges, namely the need to guarantee that unauthorized accesses do not compromise confidentiality and integrity of data and that client misbehaviors are kept under control.

Misbehaviors can be *byzantine*, i.e. erratic and arbitrary deviations from the normal behaviour, or rational, where users attempt to gain an advantage but only if their misbehaviour cannot be detected by correct clients or servers. Byzantine clients can easily be detected, since it does not attempt to hide its misbehavior, which means they can be dealt with through isolation. Therefore, the main focus is shifted towards rational clients.

Rational attacks on causal consistency consist of tampering with other replicas' operations or attacking the operation generation by attaching incorrect dependencies. To avoid such attacks the authors propose a secure form of causal consistency with the following properties:

- **Immutable history**: No operation with tampered dependencies is executed;
- **No Future Dependencies**: No operation that depends on a future operation is executed;

- **Causal Execution**: All correct replicas execute an operation respecting the dependencies defined in the operation;
- **Eventual Sibling Detection**: Given two operations with the same identifier, eventually detect and report the conflict using a fault operation;
- **Limited Omission**: A malicious replica cannot omit from the dependencies of an operation o any operation that happened before an operation in the dependencies of o.

In addition, a replica should be forced to set the real dependencies to the operations it generates, in order for the consistency to be strictly secure. Furthermore, with these restrictions it is still possible for a set of replicas to communicate through a side channel to circumvent the safety mechanisms. In order to achieve collusion tolerance, if an operation executes in an order that violates the total order, the out-of-order operation is signaled.

2.6 Databases

Finally, to finish this chapter, we investigate a selection of database solutions with the intent of studying their implementations to gain a better understanding of the desirable traits in a distributed system that deals with replicated data. Below are the options that were considered.

2.6.1 Ditto.live

Ditto [8] is a cross-platform peer-to-peer NoSQL document-based database. It is a fully distributed database that runs in the cloud and on local devices and whose nodes are able to synchronize without an internet connection through peer connectivity. It differentiates peers into Big Peers and Small Peers.

Big peers are cloud nodes and, as such, are capable of storing a large amount of data. They are capable of sharding and partitioning and follow an altruistic replication strategy, that is, they try to sync all the data it receives from Small Piers.

On the other hand, small peers are embedded in weaker machines, such as desktops or mobile devices, that do not perform sharding nor partitioning and only sync data down from nearby Small Peers or Big Peers when it has a live query. They are however, capable of employing device peer to peer communication tactics like Bluetooth Low Energy, Wi-Fi Direct, AWDL, Wi-Fi Aware, Local Area Network to transmit data.

As such, small peers are able to form wireless network meshes with other devices but only if they are sufficiently close to them. This means that for connections with more distant nodes or with nodes that are not online at the same time, simply using these small peer-to-peer capabilities is not enough. Therefore, big peers are able to bridge the gaps between disconnected meshes acting as a database capable of propagating changes across networks.

It handles concurrent updates on multiple peers without locking or consensus, by employing data types based on Delta CRDTs.

However, it does not seem to offer any kind of tool with which to handle geographic data.

2.6.2 CouchDB

CouchDB [6] is a document based database with an optimistic and lockless update mechanism. This means that if two clients try to save changes to the same document, the last to do so fails and has to retry the operation after retrieving the latest version.

It is also a peer-based distributed database that allows users to access data while disconnected by asynchronously fetching the data in remote nodes and then, when a request is made, fetching the data locally, ensuring a low latency. If changes are made to the data while disconnected they are replicated bidirectionally once the connection is reestablished.

CouchDB is designed for eventual consistency, meaning that it does not have to wait for changes to be propagated to all nodes and can instead focus on maximizing availability. It deals with concurrent updates through the usage of a Multi-Version Concurrency Control (MVCC), where as a document is updated the system saves the new version in the system alongside all the previous versions. In the event of a conflict between concurrent updates, it employs an automatic resolution mechanism.

Finally, it also allows for geographical searches and sorting by the distance to a geographical location.

2.6.3 HarperDB

HarperDB [15] is a geo-distributed database that combines SQL and NoSQL functionalities in a single hybrid model. Its main goal is to reduce latency by replicating the cloud data in an array of edge servers, so that the information is closer to the end-user and the computational load distributed, while also allowing for real-time data synchronization between nodes.

HarperDB's clustering engine replicates data between instances of HarperDB using a highly performant, bi-directional pub/sub model on a per-table basis. Data replicates asynchronously with eventual consistency across the cluster following the defined pub-/sub configuration. Individual transactions are sent in the order in which they were transacted, once received by the destination instance, they are processed in an ACID-compliant manner. Conflict resolution follows a last writer wins model based on recorded transaction time on the transaction and the timestamp on the record on the node.

It is also able to perform geographical functions including ones that return the distance to a geographical point, perform area based searches and more, with the restriction that the data must adhere to the GeoJSON standard.

2.6.4 Macrometa

The Macrometa Global Data Network (GDN) [26] consists of a geo distributed event source with a materialized view engine and CRDT based replication. As such, it is different from databases with eventual consistency, as it handles conflicts at the atomic field level and merges changes made across the network to maintain a single version of the truth via CRDTs. It allows for local reads and writes to achieve a low latency and is capable of real-time processing of streams without the need of sending the data to the cloud.

It integrates document databases, graph databases, Key-Value, Pub-Sub Streams, etc. in a single layered concept and decouples its data storage technology from its data model. All client requests to nodes inside the same data center return the same view on the data.

GDN is also composed of fabrics which are collections of edge data centers linked together. Different fabrics are usually used for geo-fencing setups, as the data inside them is isolated from one another.

Macrometa also allows for geographical querying by also adhering to the GeoJSON standard and is therefore able to support polygon-based, distance and other types of queries.

2.6.5 Akka Serverless

Akka Serverless [1] is a Platform-as-a-Service that handles the configuration and management of databases, servers and clusters in a database-less programming model and serverless runtime. It is API-first, meaning that developers create the data objects that make sense for their service without needing to know how that data needs to be persisted in the durable storage.

It requires the developer to choose the state model for the data storage and management of entities. Value Entities are Key/Value stores and offer the traditional approach to persistence-storing state as a single, complete unit; Event Sourced Entities store each update as an event, which together build the state of the entity and Replicated Entities distribute state using a conflict-free replicated data type (CRDT) which shares data across multiple instances. However, regardless of the model, I manages concurrent accesses to the same entity sequentially, which can sometimes act as a bottleneck.

It also does not seem to offer any geographical data management tools.

2.6.6 SkySpark

SkySpark [50] is an analytics tool that automatically analyzes data from smart devices and equipment systems to identify issues, enabling improved performance, reduced downtime, and operational savings. It is based around a Time Series database designed for IoT data that stores and compresses heterogeneous data and creates unified models around it.

It is then able to apply an analytic engine that automatically processes rules and algorithms on your data to find Sparks — things that matter, as well as being able to visually present the data and its analytic results.

2.6.7 Cloud Firestore (Firebase)

Cloud Firestore [4] is a NoSQL document-based cloud database hosted by Google for mobile, web, and server development. It uses data synchronization to update data on any connected device and is able to provide offline support through local caching. It provides automatic multi-region data replication, atomic batch operations, and real transaction support, as well as extra Google Cloud features like Cloud Functions. However, it separates itself from the rest in the sense that it provides strong consistency guarantees. Finally, it also offers geographic querying support through a *Geohash* system.

Table 2.1: Database functionalities

Functionality	**Ditto**	**CouchDB**	**HarperDB**	**Macrometa**	**Akka**	**SkySpark**	**Firebase**
Peer-to-peer capabilities	✓	✓					
Geographical query tools		✓	✓	✓			✓
CRDT-based conflict resolution	✓			✓	✓		
Analytics tools						✓	
Offline support	✓	✓					✓

2.6.8 Comparison

All of the database solutions above provide some sort of non-relational storage and retrieval of data. Moreover, despite each possessing its own set of intricacies, some share common strategies to address the challenges they are faced with. For example, both Ditto and CouchDB rely on peer-to-peer connectivity to ensure high availability, whereas most other databases communicate through client-server protocols with the cloud.

Furthermore, most of the databases are designed for eventual consistency, as another way of guaranteeing high availability. Ditto, Macrometa and Akka also resort to CRDTs as a means for conflict resolution between data replicas. However, Google's Cloud Firebase chooses instead to provide strong consistency guarantees and utilizes cache storage to improve the availability of the system, even when offline.

Finally, only some of the implementations were equipped with geographical querying tools, those being CouchDB, HarperDB, Macrometa and Cloud Firebase. The table 2.1 further illustrates the comparison between the databases.

2.6.9 Time-series databases

Finally, it is also sensible to elaborate about the concept of time-series databases, since the focus of this thesis is similar to this type of database, albeit through different means.

When dealing with large amounts of time-stamped data from sources such as IoT devices, financial transactions, or sensor networks, time-series databases can provide highly efficient solutions. InfluxDB [17] is a distributed time-series database that uses a SQL-like query language and supports data retention policies for data expiration. TimescaleDB [54] is built on top of PostgreSQL and provides automated data chunking and compression for efficient storage and retrieval. Graphite [14] stores data hierarchically and supports various data sources and visualisations. Additionally, Prometheus [39] is designed for monitoring and alerting, supporting data scraping, aggregation, and querying.

Time-series databases offer optimised performance for data with a time attribute, whereas, in our work, we try to achieve similar benefit for data with a location attribute. Furthermore, FocusDB focuses on adapting the consistency offered to different clients based on interests provided by the user regarding specific locations. This allows applications to exploit better a trade-off between consistency and performance based on application-specific logic.

3

System Design

This chapter describes the overall design of the system developed for this thesis work. We start by providing a general understanding of the goals behind the design choices, followed by delving into the characteristics of the architecture of the system, as well as each of its main components and features.

3.1 System Overview

FocusDB is a data management system with the purpose of supporting mobile applications with efficient data access. It allows developers to define the data model for their applications, as well as the detail and structure of the data that is received by the clients.

The system architecture, represented in figure 3.1, consists of a single centralized server that is able to communicate with several mobile clients over a network. The server, which stores all of the application data, responds to calls made by the clients through the FocusDB API, where clients can read and update the data records that the server holds. Behind the scenes, the server organizes the data into several data views that are kept constantly updated and consistent, and that serve the purposes of providing efficient data access and facilitating the management of geographically-based queries.

In FocusDB, all data objects have a location property, in order for the system to be able to identify the real-world location that is associated with them. Besides location, data objects can hold any number of other different attributes, according to the needs of the application.

However, it may be the case that some of the data attributes only start becoming relevant to the end-users once a certain proximity to the data location is reached. For example, in the case of an app that helps the user find parking spaces, a user may not be interested in the specific location of several parking spaces if they are quite far away from the user, instead preferring to be shown only the number of spaces in that area. Because of this, FocusDB allows an application to filter the data object attributes before they are sent over to the client, based on the client and data locations and through the choice of a specific data view. This allows for each data object to be exposed to the client in multiple

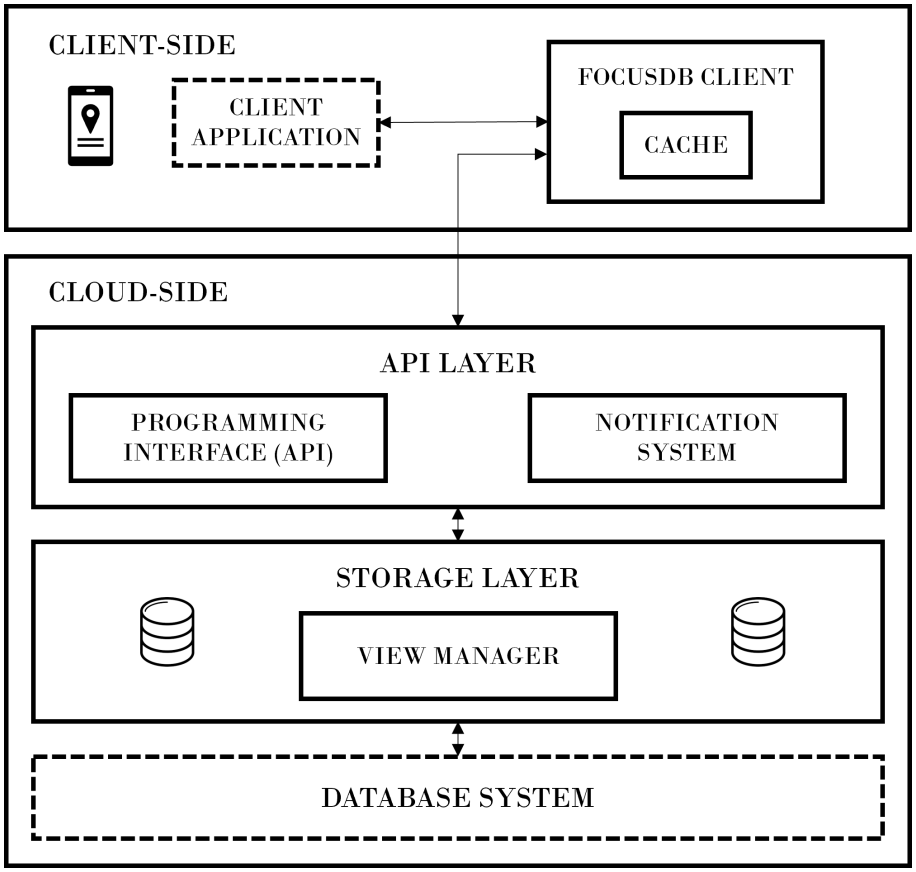

Figure 3.1: FocusDB System Architecture

representations, each holding a different level of detail.

3.2 Cloud-side Architecture

On the cloud-side, the system can be be divided into two distinct components:

API Layer: This layer is responsible for exposing the access points for data access and manipulation, serving as an intermediary between the clients and the database. It is tasked with performing the translation of client requests into database queries, executing the logic of requests, and proactively notifying clients about changes relevant to the clients' cached data.

It holds a set of endpoints designed for managing the underlying data collections and materialized views, and another set for accessing and manipulating the data items. Furthermore, it is also endowed with a subscription-based notification system, that ensures that the data is kept up-to-date on the client-side.

Storage Layer: The storage layer is responsible for providing durability to the system's data, by managing the storage of its data collections. It also builds a set of materialized

views, which are based on the data attributes and consist of unique data objects or aggregations of objects that are grouped by a specific attribute. This allows us to display different levels of detail for each data object.

This component relies upon the usage of a database management system (DBMS), the choice of which can be adapted according to the developers' needs, since the FocusDB API layer can be easily adapted for different solutions. However, since the system relies heavily on the aforementioned materialized views, it is of the utmost importance that those be kept up to date in regards to the main data collections. Therefore, in the event of the chosen DBMS not implementing a mechanism for incremental management of materialized views, the FocusDB View Manager can be used instead, so that the system can perform its function as designed. In the case of our prototype, MongoDB was chosen as the database system, which does not perform incremental management of materialized views, meaning that the FocusDB View Manager had to be utilized.

3.3 Client-side Architecture

On the client-side, there is a single FocusDB component:

FocusDB Client: The FocusDB client allows the application running in the mobile device to interact with the server-side components in a transparent and easy way. The client exposes a client library to allow for communication with the server, by submitting queries and updates and receiving change notifications. Furthermore, it also holds data received from the server in a local cache in order to support immediate replies to applications and to reduce the load on the central component.

The interaction between the FocusDB client and the server-side infrastructure is the following. In the case of a read operation, the client sends a data request to the API, which communicates with the storage layer to retrieve the relevant data. The API then returns the data to the client, which stores it in its local cache. If the user has a continued interest in the data it just received, the client can subscribe to the server notification system, so that the server sends new data whenever a write operation modifies, adds, or removes data relevant to that request.

Conversely, in the case of a write operation, the client submits its request to the API, which contacts the storage layer to reflect the change in the data layers. If the storage layer DBMS cannot update the materialized views, the FocusDB View Manager steps in to update the relevant data entries in the views, therefore keeping them consistent with the information in the data collections.

3.4 Data Model

In order to allow for the system to be adaptable and flexible to the most diverse data formats, it is necessary to define a generic data model over which the database reading

and writing operations can be performed. As such, the following structure defines the organization of data inside the system's storage layer.

FocusDB acts as a key/value store, with the main advantage of being able to provide different versions of the same data object at different consistency levels. The main concepts that constitute the data model are detailed as follows.

Objects represent individual data elements stored in the system, which are assigned a unique identifier, stored in collections with a flexible schema and can hold a multitude of different attributes, depending on the needs of the application. We separate objects into two categories: *Base Data* and *Derived Data*. Base Data comprises the objects that are written directly by the application, which represent the unaltered version the data. Derived data, on the other hand, encompasses all other objects that have been obtained through the transformation of other Base Data or Derived Data objects. These transformations can be as simple as reducing the visibility of specific attributes or as complex as combining features from multiple objects using operations such as union, intersect, and general aggregations. The results of these computations are stored in materialized views, and can be used for further operations.

Collections are sets of data that can be comprised of Base Data Objects or Derived Data Objects. Collections that are the result of the application of a view, are called materialized views and are made up of the Derived Data objects that result of that operation.

Views are defined as a function that can be applied on data collections or other views. They are built over the global data collection, which means that their definition is global within the system. The definition of a view is kept in an object that, among other attributes, holds the view unique identifier, the collection or view it is to be applied on, and, most importantly, a function that represents the query used to compute that view. The main purpose of views is that it allows the filtering of data fidelity, in order to leverage the geographic properties explained earlier in section 1.2. To avoid having to compute one or more views every time a data request is received, the views are materialized server-side, so that the process of data fetching is expedited.

Views are also defined using an attribute of the underlying collection or view as the key for that view. As such, each object in the original collection will be associated with the entry that corresponds to the value of the attribute in the object. For example, an object where the attribute 'color' has the value 'blue' will be associated with the entry 'blue' in a view based on that attribute.

Each materialized view entry can be composed of Base Data objects, if no transformation operation is applied to them, or Derived Data objects with possibly a lower fidelity level than its Base Data counterparts. The reason behind this design choice is due to the manner in which the data requests will be performed. Users will be interested in data items belonging to a certain area or in the vicinity of a given geographic position. Therefore, it stands to reason that associating the data objects to attributes that describe their location, such as *county*, *district* or *neighbourhood* can be very beneficial for system performance, since all relevant data items will already be bundled together. Consequently,

users can query the materialized views with these attribute values in mind, instead of using coordinate-based queries, that is, queries that describe a geographical area using coordinates, and for which the database system has to scan the entire data collection in order to return the objects whose locations are contained in said area.

Regarding reading and writing operations, the system supports the fetching of both Base Data and Derived Data objects from materialized views. However, when it comes to writing, clients are not allowed to perform updates to the materialized views themselves. Instead, they must operate on the full representation of the Base Data objects in the main collection, whereby the server will then ensure those modifications are put into effect in the respective materialized views.

3.5 FocusDB Components

We will now explain some of the design choices behind each of the main FocusDB components. We won't delve very far into implementation details, since those will be further explored in the upcoming section 4.

3.5.1 Programming Interface (API)

The Data API is one of the two main components of the API layer and is responsible for exposing the access points for the system. At this level, two categories of operations are defined: those to be performed by the applications for managing collections and views, and the remaining belonging to the client interaction with actual data. The first is comprised of the endpoints for view and collection management, whereas the second encompasses all read and write operations (*inserts*, *updates*, *deletes*, and *gets*) for the data objects.

The following endpoints pertain to the management of collections and views:

- **Create Collection**: Creates a new collection in the storage layer with the given name;
- **Delete Collection**: Deletes an existing collection with the given name from the storage layer as well as all its contents;
- **Get Collection Names**: Lists all collection names in the system.
- **Create View**: Introduces a new view definition in the system and materializes said view. A view definition must detail the view's name and type, that together form its unique identifier, the collection that it is to be applied upon and the attribute that will act as key, and finally the transformation to be applied to the data objects. An example of a view definition is given in listing 1.
- **Delete View**: Deletes the view definition with the given type and name as well as the materialized view and all its entries.

- **Get All View Definitions**: Lists all view definitions in the system.
- **Get View Definition**: Returns the details of the specific view definition with the given type and name.

The reading and writing endpoints are the following:

- **Insert Base Data Object**: Inserts the new data object in the main collection as Base Data.
- **Update Base Data Object**: Updates an existing data object in the main collection to display the requested changes.
- **Delete Base Data Object**: Deletes an existing data object from the system by removing it from the main collection.
- **Get Data**: Returns data from a materialized view, according to a given query.

  The query can take one of two types. It can be of type *Attribute*, meaning that the user wants to query the view based on the attribute that acts as key, and where the returned data will consist only of data associated with that attribute value, or it can be of type *Location*, where the user provides a geographical position and a max radius (and optional minimum radius), to which the system will query the entire view to return all objects that are contained in the described area.

  The view selected in the query also alludes to an implicit choice of the level of data detail of the reply. It is the responsibility of the application using the FocusDB client to select the appropriate level of detail of the requests, and of the application designer to define a pre-determined set of levels of detail materialized as views.

The collection and view management endpoints were not designed to be used by common system users, only by system administrators. A method of role-based access control to the API has not been implemented in this work. Similarly, we also do not consider operations issued by malicious clients, leaving both these points for future iterations.

3.5.2 Notification System

The second of the two components of the API Layer is the Notification System. This component is responsible for ensuring that the data residing on the client-side, i.e. the FocusDB client, does not become outdated in relation to the data maintained in the centralised component. When a client requests data through the API, most of the times it is declaring that it is interested in that set of data and, as such, would like for it to be kept up to date over a period of time. We will refer to this set of data as the client's *Interest Set* from here on and it is similar to continuous queries in database systems [3].

One could place the responsibility of keeping the cached data updated on the client-side, by stating that it should repeat the same data request periodically, in order to fetch

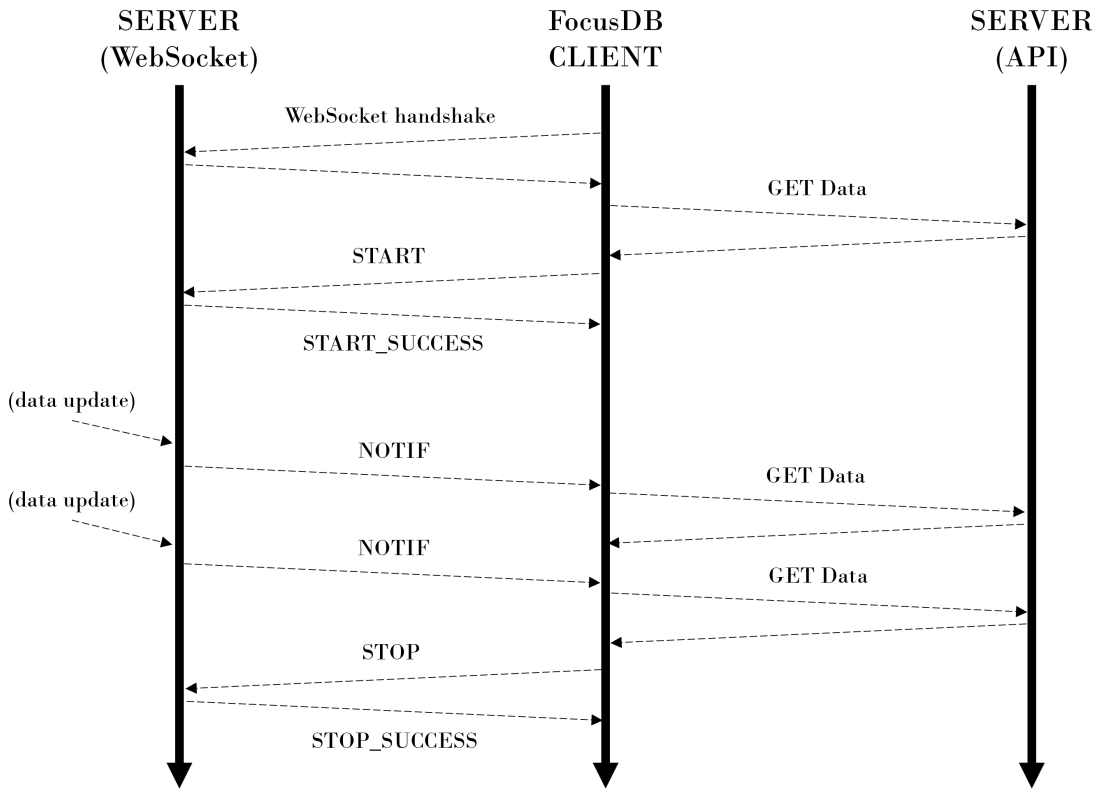

Figure 3.2: FocusDB Notification System Protocol.

any modifications on that dataset. However, it is not hard to envision that such a strategy would prove itself inefficient, due to the fact that the client does not possess the capability of knowing when its interest set data has changed. Because of this, many of the repeated data requests can result in the return of the exact same set of data the client already holds, especially in cases where the underlying data is not updated quickly. This ultimately, would incur an unnecessary usage of system resources as well as network bandwidth.

In order to avoid this issue, FocusDB instead relies on a notification system that informs clients whenever any relevant updates happen to the data that is cached by them and belongs to their interest set.

The system works as follows: At any time, the client can contact the centralized component through its WebSocket-based Notification Protocol, which is represented in figure 3.2, and start the flow of notifications. The client needs only, in the start of the communication, to inform the server what the data request is for the set of data it is interested in. From that point on, the client will be subscribed to the corresponding interest set and will begin receiving notifications whenever a change in the data occurs server-side, along with all other subscribed clients. This notification acts as an invalidation report, with the client knowing that it must repeat the data request to receive the updated data. At any point, the client can unsubscribe from the interest set or subscribe to multiple interest

sets simultaneously.

The notifications in the protocol only inform that new data is available, instead of carrying that data immediately to the clients. This is because, in the future, it will allow for the client to dictate when the repetition of the data requests should be done. Instead of performing a data request for every notification received, the client could be able to delay the data fetching until a certain time interval has elapsed or until a certain number of notifications have been received by the user, allowing it to reduce the traffic even further. This would correspond to a mechanism of data consistency control dictated by the needs of the client, especially useful for more distant data items where the need to receive updated results is not as pressing as with more relevant data. Despite this mechanism not yet being implemented, the design choices behind the notification protocol aim to facilitate such an effort [58, 42].

3.5.3 Incremental View Manager

The final of the main FocusDB components is the Incremental View Manager. Located in the storage layer of the server, the view manager is tasked with keeping the materialized views updated in relation to the main data collection. This is especially important in the context of our system, due to the fact that read operations are solely performed over materialized views. Therefore, it is imperative that those collections are kept updated so that clients can receive the latest version of the data items.

However, not all database management systems possess the capability of performing incremental management of materialised views, and those that do, do not take location properties into consideration to govern this process [5]. In systems that lack such a mechanism, the entries of each view are instantiated based on the collection contents at that moment. That is the case of MongoDB, the database our prototype uses as the storage backend. As the base collection suffers modifications, the views remain static and are not able to reflect the current state of the data unless the views are manually modified or recomputed entirely. It is necessary, therefore, to address the possible lack of such a component in order for the system to operate properly.

The FocusDB implementation of the incremental view manager consists of a component that reacts to updates made on the data collections and reflects those changes in the various materialized views. When a Base Data object is created, modified or removed from a collection, the view manager will first process the modifications in order to determine the views that need to be operated on and in what capacity. It then queues the modifications, groups them per view and then executes the updates in batches, so that the process can be as swift as possible. Furthermore, for each of the updates, instead of recomputing the entire view, an action that can often be very computationally expensive, the incremental view manager affects only the portion of the view that needs to be updated, and then stores the modified results in the database.

Despite the aforementioned efforts to achieve greater efficiency in the update process,

the changes on the materialized views still need to be conducted asynchronously to the API, since the update process is time-consuming. This has the consequence of introducing slight inconsistencies into the system, while the updating process is ongoing. Despite this, considering that amount of inconsistency generated is quite small, and that the benefits of this component are crucial to the operation of the entire system, this small amount of inconsistency is considered to be acceptable.

3.6 Final Remarks

In this chapter, we presented the overall design of the FocusDB system, and detailed each of its main components: the Data Model, API, Notification System and Incremental View Manager.

We believe this design to be effective in addressing some of the shortcomings of modern distributed solutions with regards to the handling of geographically associated data, as detailed in section 1.2. The upcoming chapter 5 will attempt to support this claim by comparing the performance of FocusDB to the previous solutions in a simulated scenario that mimics a real-world use-case. However, first we will explain some of the implementation choices in order to give a more comprehensive view into the system.

4

Implementation

In this chapter, we intend to delve deeper into the implementation details of the FocusDB prototype. Going from component to component, the goal is to expose some of the intricacies of the development efforts, as well as some of the challenges that emerged during it, and how those were handled.

4.1 Supporting Technologies

The development of FocusDB first started by implementing a server that exposed a simple API capable of accessing and modifying generic data, i.e. data with no predetermined set of attributes. The programming language chosen for this endeavor was Java, specifically Java 1.8, due to the fact of it being a robust language with an extensive library of tools designed to aid in the development of distributed systems.

The server consists of a Jersey JDK HTTP server, over which a set of resources are implemented, which are designed to handle the system's API calls. We experimented with different server implementation choices, such as Jetty or Tomcat, but noticed that the system performance was much higher with the basic Jersey implementation than with the more complex options. Therefore, we chose that to be the base server implementation for our prototype.

As for the storage layer, we chose MongoDB [29] as the database management system. The case could have been made for many other systems, since the only requirements imposed by FocusDB is that the database offers a flexible schema and is capable of implementing our definition of data collections and materialized views. The flexible schema allows us to have more freedom in the attributes each data object can hold, and frees the database system from having to ensure all data follows the same structure. MongoDB adheres to these requirements, and we utilize it to implement FocusDB according to our needs, taking advantage of some of its functionalities, such as its aggregation capabilities or the ability to perform geospatial queries.

4.2 Data Model

With regards to the Data Model, there are a few details that are important to be mentioned. The first one is related to how the geographical data is stored within the collections of data. The way MongoDB processes queries imposes that the geospatial data be stored in a specific JSON format, named GeoJSON [30, 10]. Although is was possible to also impose that restriction on the user data, we felt it more sensible to simplify the process of data insertion to the user. As such, to insert a data object into a collection, the only required attributes are the object ID (*id*), the latitude coordinate of the object's position (*lat*), and the longitude coordinate (*lng*). When an object is about to be inserted into the collection, the system translates the set of coordinates into the GeoJSON-compliant format and associates it with a new attribute *location* in the object's document.

Another relevant point has to do with the way in which the system stores data on materialized views. As previously stated, each materialized view is associated to a specific view definition, where each view definition holds a number of attributes that are essential to the creation of its materialized view, as well as the access to its data. These full attribute set is listed as follows:

- *type* and *name*: Together, these two attributes form the unique identifier for the view. Although not enforced in the implementation, the developer can consider view types and names as categories to group certain views together. During the experimental phase, we considered type to be the attribute the view is based upon and name to describe the level of detail exposed by the view objects. For example, type *Streets* would categorize a view organized by street designation, and name *nSpots* would describe the detail level as the count of the number of objects associated with each street. Finally, *Streets_nSpots* would be the resulting view identifier.

- *collection*: The unique identifier of the collection over which the view is to be applied on.

- *attribute*: The attribute present in the main data collection that is to serve as key for the materialized view.

- *pipeline*: This attribute describes the operations that MongoDB must perform in order to alter the level of fidelity of the data in the view. The transformations are performed through the use of the aggregation functionality in MongoDB, where it is required that the user describe the pipeline of operations to be performed, as well as their order. A full listing of the possible operations is detailed in the MongoDB manual [27, 28], and an example is provided in the *Create View* endpoint explanation in the following subsection 4.4.1.

- *centroidCollection*: This is the name of the collection that contains the centroids for the entries of that view. Its purpose will become clearer in the following chapters,

but it allows to perform location-based queries on the materialized views, without the use of an explicit attribute value. If not specified at the moment of view creation, the value of this property is set to the default value: the prefix *CENTROID* concatenated to the main collection name and to the view's key attribute.

As was previously stated, the materialized views are constructed in such a way that a specific attribute will act as the key for that view. Each entry will be associated to a value the attribute takes in the main collection, and will contain some representation of all data objects that contain that value. The fidelity of the representation can range from exhibiting the exact same detail as the Base Data objects, to only some of the original attributes, or even to the result of an aggregation, such as the count of the number of objects with that attribute value.

4.3 Geographical Querying Mechanism

These aforementioned characteristics, however, pose a challenge to the geographical querying mechanism. The *Get Data* endpoint, first mentioned in subsection 3.5.1, has two possible query types: attribute and location. These are further detailed below in the next subsection 4.4.2, however succinctly, the attribute data request type consists of selecting the data objects based on one or more view key attribute values, whereas the location type consists of the client detailing a geographic area where the data items it desires are located. For type attribute, there is no problem with the querying; the system simply returns the view entries corresponding to the query provided in the request.

However, with type location the situation is more complex, partially because of limitations on the database system side and partially due to the nature of the data. On one hand, MongoDB is unable to perform geographic queries on objects nested in an array inside each of the view entries. On the other, sometimes the reduction of fidelity of the items removes their geographic position attributes, for example, in the case of an aggregation where it is only relevant to know the number of objects in a given area. In these situations, it is necessary to assign each of the materialized view entries with a representation of their location. In our prototype, the solution devised consists of taking the positions of all data objects associated with each view entry at the base fidelity level, and calculating their average position, which is named the *centroid*. These centroids are stored in separate collections and are associated to their corresponding entry in the views. Views that use the same key attribute, will also use the same centroid collection, since each centroid based on the objects with the same key attribute value. Finally, each of the views is geographically indexed with a MongoDB *2dsphere* index [31] on the centroid object, to allow for geographical queries to be able to calculate geometries, using a spherical earth-like shape. It is also possible for the developers to manually create the aforementioned centroid collections, and add to them more detailed GeoJSON geometries, such as lines or polygons, allowing for more exact data responses.

This implementation, however, has one relevant consequence, which is that, since the positions are associated with the view entry and not to the objects themselves, all objects will end up being returned to the client, even if not all of them intersect the area described in the geographical request. Usually, each view entry document tends to not hold a large number of objects, so extra filtering of the results can usually be done on the client application side. If this is not the case, we recommend splitting the data into separate entries to improve performance.

4.4 Programming Interface (API)

The FocusDB programming interface, or API as it is more commonly referred to, is a crucial system component, responsible for exposing the access points to the clients. It enables clients not only to be able to retrieve and manipulate data, but also to have a means of controlling the system in accordance to their needs.

The API is comprised of two layers of components: JAX-RS Resources that capture and respond to HTTP requests, and their corresponding Data Layer Objects, that deal with the logic behind every request.

JAX-RS is used to represent RESTful resources through the means of plain Java classes [18]. By providing specific annotations, it enables the implementation of resource methods that are capable of answering different types of HTTP requests, according to REST standards. Root resource classes are "plain old Java objects"(POJOs) that are either annotated with *@Path* or have at least one method annotated with *@Path* or a request method designator, such as *@GET*, *@PUT*, *@POST*, or *@DELETE*.

In the FocusDB API, there are two main resources: the Control Resource, responsible for all endpoints concerning the management of collections and views, and the Data Resource, which contains all reading and writing operations for the data objects. Besides these two main resources, FocusDB also exposes other minor resources, that mostly pertain to the evaluation of the system for this thesis. Each resource is connected to its corresponding Data Layer Object, which acts as an intermediary between the resources and the storage layer/other system components, by parsing the client requests into the relevant component operations, such as database queries, as well as dealing with the necessary logic behind each API call. To translate the client requests into MongoDB operations, the Data Layer Objects perform calls to the API exposed by the MongoDB Java client, which exposes a set of methods that enable read and write operations on the collections held in the database, such as *find*, *insertOne*, *deleteOne* or *updateOne*.

The following subsections present the complete list of resources and their corresponding endpoints, the majority of which were listed in section 3.5.1, but will now be described in greater detail, as to provide better insight into their inner workings.

4.4.1 Control Resource

As previously mentioned, the scope of the Control Resource pertains to all aspects regarding the management of data collections and views. To help with the interaction with the storage layer, the control resource shifts those duties to the Control Data Layer, where the bulk of the computing is performed. The endpoints pertaining to the Control Resource, whose path is always prefixed by */settings*, are the following:

Create Collection: POST /settings/data/{name}

The Create Collection endpoint is a HTTP POST request that receives a single path parameter, corresponding to the name to be given to the collection that is to be created. The data layer first verifies whether a collection with the given name already exists, launching an exception if that is the case. If it is not, then the server creates a new entry in the metadata collection of collection names and creates the collection in the database.

Delete Collection: DELETE /settings/data/{name}

Delete Collection also receives a collection name as a path parameter, which the data layer uses to delete the collection, its associated data objects, as well as the corresponding metadata. Furthermore, since views are associated with an underlying data collection, when a collection is removed, the system also deletes all views that are based on it.

Get Collection Names: GET /settings/data

The final collection management endpoint is Get Collection Names, a GET request that returns all collection names, as present in the corresponding metadata collection.

Create View: POST /settings/view

The Create View endpoint requires the client to submit a JSON object in the request, containing the various parameters necessary for establishing a new view: the view's *type* and *name*, the *collection* to apply the view on, the collection *attribute* to use as key for the view, and finally, the *pipeline* of MongoDB transformations to apply on the data objects. In the event of the user not providing a pipeline, no modifications are done to the data objects, therefore creating a view with the base fidelity level.

The process begins with the parsing and validation of the submitted JSON object, returning an error if the JSON is poorly formatted, if an essential property is missing (type, name, collection and attribute), if there is already a view with the same type and name, if there is no such collection, or if there is no such attribute in said collection.

Next, the new view is assigned a centroid collection. If the collection does not exist, it is created and the centroids for each of the view's attribute values are calculated. After that is concluded, the data layer then requests an aggregation from MongoDB on the specified collection with the provided pipeline, where the data object transformations taken into effect, and the centroids are associated with each entry. Finally, the view is

```
{
    "type": "Streets",
    "name": "Full-Detail",
    "collection": "spots",
    "attribute": "street",
    "pipeline": "[
                    { $match: {"free": true} },
                    { $group : {"spaces": {"$push": "$$ROOT"}} }
                ]"

}
```

Listing 1: Create View request JSON with full fidelity.

geographically indexed with a MongoDB *2dsphere* index on the centroid objects and a view manager thread is started to handle any upcoming data modifications. The view is from this point on ready to be accessed for its data, with the view manager automatically taking care of keeping it up to date with its main collection.

Listing 1 presents an example of a Create View request JSON object. It is meant to create view 'Streets_Full-Detail' over the collection 'spots' and attribute 'street'. Its MongoDB pipeline describes two operations. First, the *match* stage, similarly to a SELECT operation in SQL, dictates that only objects with the attribute 'free' that equal 'true' shall be part of the materialized view. And secondly, the *group* stage, which states that all view entries shall contain an array named 'spaces', containing all data objects as they appear in collection 'spots'. The result is a view keyed by attribute 'street' and whose objects expose the same level of fidelity as the base data objects. Conversely, listing 2 presents a pipeline where the group stage performs a *count* operation instead of the previous *push*. Consequently, the view objects will contain only a count of the objects in the main collection where the 'free' attribute is 'true', corresponding to a decrease in the fidelity level.

Delete View: DELETE /settings/view/{type}/{name}

Delete View simply takes the type and name of the view to be deleted from its two path parameters and orders its deletion from the database. The view is deleted and its corresponding view manager thread is stopped. If after the deletion, no other view is based on the same collection and attribute, the centroid collection is no longer necessary and is therefore also deleted. Finally, in the event of the view type and name initially by he user provided not corresponding to a view ID in the system, the endpoint returns an error.

List View Definitions: GET /settings/views

```
{
    "type": "Streets",
    "name": "Count",
    "collection": "spots",
    "attribute": "street",
    "pipeline": "[
                    { $match: {"free": true} },
                    { $group : {"count": {"$count": {} }} }
                ]"

}
```

Listing 2: Create View request JSON with aggregation.

This endpoint returns all of the view definitions contained in the system's view metadata collection. It returns a list of objects containing all of the attributes pertaining to the definition of each view.

Get View Definition: GET /settings/view/{type}/{name}

Similar to the previous endpoint, but instead of returning a list, this endpoint return only the view definition object referring to the view with the provided type and name.

Recalculate Materialized View: POST /settings/view/{type}/{name}

Due to the existence of the system's view manager, the utilization of this endpoint is not required during the normal operation of FocusDB. It can be used, however to manually request the recomputation of a materialized view, given its type and name.

Get View Contents: GET /settings/view/data/{type}/{name}

Get View Contents is an endpoint designed for debug purposes only, since it returns the entire contents of a single view. Proper view content access is done through the Data Resource endpoint Get Data.

4.4.2 Data Resource

Whereas the Control Resource was designed to be used by developers and system administrators, the Data Resource is instead responsible for communicating with clients and act on their data-based requests. The system implements the FocusDB Data Layer to serve as the intermediary between the resource and storage layer for the execution of data writes and reads. The endpoints that comprise the Data Resource are the following:

Insert Data Object: POST /api/data/{name}

This data endpoint deals with the insertion of a new data object in a given collection, according to the collection name that is passed as a path parameter. The client must provide the data object in JSON format, which is required to contain at least three attributes: the object's unique identifier, the latitude value associated with its real world position as well as the longitude value. If the JSON is poorly formatted, missing key attributes, using a repeated identifier or trying to be inserted in a collection that does not exist, the endpoint will return an error. If the object clears this validation step, its latitude and longitude values are translated into a *Point* type object in the GeoJSON standard, and the object is then inserted in the corresponding collection. As a final step, the View Manager is notified on the insertion, which will trigger an update on the materialized views associated with that collection.

Update Data Object: PUT /api/data/{name}

The Update Data Object is designed to update already existing data objects on a given collection. Similarly to the Insert, this endpoint receives the collection name as a path parameter and consumes the data object JSON. If the JSON is poorly formatted, missing the object ID attribute, trying to perform an update on an object that does not exist or in a collection that does not exist, the endpoint will return an error. If none of these conditions are met, then the update proceeds. The latitude and longitude parameters, if present, are translated into GeoJSON and the update is performed on the selected object. Finally, the View Manager is triggered for the update operation.

Delete Data Object: DELETE /api/data/{name}/{id}

The last data writing endpoint is the Delete Data Object endpoint, which is responsible for removing a given data object from its collection. The collection name is given as a path parameter, as well as the object ID. If the collection exists, the order is given to the database to delete the object from the collection. It is not necessary to verify whether the object exists before attempting a deletion. However, we must check if a deletion was indeed conducted by the database, since it is necessary for the alert to the View Manager.

Get Data: POST /api/data

Finally, the data reading endpoint for the FocusDB system is the Get Data endpoint. Instead of a GET operation, the interaction method with this endpoint is a POST operation. This is due to the fact that the requests made to this endpoint possess many different parameters, making it impossible to implement this method using only a GET annotation along with path and query parameters.

The Get Data request is mainly composed of three parameters: the view *type*, view *name* and request *condition*. The view name and type constitute the identifier of the view the client wished to receive data from. This choice is place on the side of the client, which means that the application must know which views exist in the system, as well as the

```
{
    "viewType": "Streets",
    "viewName": "Count",
    "condition": {
      "type": "attribute",
      "query": "{ '_id': 'Avenida Fontes Pereira de Melo' }"
    }
}
```

Listing 3: Get Data request JSON of type attributte.

```
{
    "viewType": "Streets",
    "viewName": "Count",
    "condition": {
      "type": "location",
      "lat": 38.72936403753,
      "lng": -9.14793017004,
      "max": 200
    }
}
```

Listing 4: Get Data request JSON of type location.

level of fidelity of the objects they hold. Only by having this notion, can the applications increase or decrease the level of detail in the data exposed to the user.

The final request parameter, the condition, is a JSON object that, in essence, describes the query to be performed on the selected materialized view. It can be of two types: *attribute* and *location*. Condition objects that are attribute-typed hold only one other parameter, the predicate for a MongoDB match query, written with the MongoDB operators. This query is to be made in an attempt to match the values of the attribute serving as key to the view. The request in listing 3 exemplifies a request for all objects that are associated with the value 'Avenida Fontes Pereira de Melo' in view 'Streets_Count'. This query is then applied directly to the materialized view, and the matching records are returned.

On the other hand, location conditions are geographical queries, describing a circle or a ring that encompasses the area the client wishes to receive data from. The request object must describe the center of the circle through its latitude and longitude coordinates, as well as the max radius of the circle, along with an optional minimum radius. This query is only possible because of the centroid representation of each view entry, enabling the key attribute values to demonstrate a real-world position. The centroids that are present inside the area described in the request will correspond to the view entries that are returned to the user. As previously mentioned, since this method returns objects based

on the positions of the centroids and not the objects themselves, there is a possibility that a request returns objects that do not fall within the area described, in the case that a centroid is in the area but near its edge. The contrary is also possible if the centroid falls just outside the area, with some objects landing inside.

If the user request data from a view that does not exist or submits an object that does not adhere to the aforementioned request structure, the endpoint will return an error instead.

4.4.3 MongoDB Resource

The resources listed from this point on serve only for evaluation purposes, but since they were important in the development efforts of this thesis, they will be detailed nonetheless. The MongoDB resource is a debug and experiment oriented class and offers direct communication with the MongoDB instance.

Drop Database: GET /mongo/drop

This endpoint, together with Seed Database, served the purpose of resetting the database state to its initial state. It is crucial that the database begins in the same state when repeating experiments with the same testing parameters, in order to guarantee consistency in the ensuing results.To ensure this happens, the first step is to drop every collection from the database, which results in the removal of all data objects from the database.

Seed Database: GET /mongo/seed

Seed Database is the final step of resetting the database. After the database has been dropped, the first step is to recreate the collections. After this, a file containing the complete testing dataset, which is present within the server machine, is loaded onto the server instance. As it is loaded, the MongoDB Data Layer, the data layer responsible for the operations in this resource, inserts the data in the respective collections in batches of 100000 data objects, since that is the limit imposed by the MongoDB instance. Finally, after the data has been successfully reinserted, the last step is to redefine and recreate the materialized views. When this is complete, the server has returned to its original state. This endpoint must only be called after having dropped the existing database, or it will not be able to return the database to its initial state.

Get Data Object By Geographic Position: POST /mongo/data/object

Besides resetting the database, the MongoDB resource implements an endpoint that enables the fetching of a single data item from a given collection, according to its latitude and longitude values. This is because, during experiments, the process of building some of the data requests required knowledge of the values of the key attribute. Hence, this endpoint was developed in order to fetch the nearest object to this location directly from

the database. The endpoint takes a JSON object that contains the collection name, latitude and longitude values and returns the nearest object up to ten meters away from the point provided.

4.4.4 Ghost Clients Resource

Finally, Ghost Clients is a resource that is only relevant in the evaluation portion of this thesis work. The purpose of this resource in the experimental procedures is to simulate clients performing updates on the data collections, in order to add entropy to the data. This is vital for a realistic and thorough evaluation of FocusDB, since real world situations are not static datasets that remain constant throughout a long period of time. Instead, by updating the data objects periodically, the system is able to be tested in a way that more closely resembles reality, which ensures that the validity of the results obtained.

The Ghost Clients resource employs a Data Layer that launches a number of Ghost Clients, which are capable of randomly selecting a number of data objects in the database and update one of their attributes. The endpoints listed below control when the ghost clients start and stop operating.

Start Ghost Clients: GET /ghostclients/start

Starts the Ghost Clients functionality by providing two query parameters: the number of ghost clients, and the rate of the changes in the data collection. The data layer launches the number of clients specified in the request, which perform availability changes according to the request change rate. The data changes consist of selecting a random object in the database and updating one of its attributes through the Update Data Object endpoint of the Data Resource.

Stop Ghost Clients GET /ghostclients/stop

Stops the threads running the ghost clients.

Get Ghost Clients Status /ghostclients/status

Returns whether there are ghost clients operating on the database at a given moment.

4.5 Incremental View Manager

This next section will delve into the implementation details of the Incremental View Manager, an essential component to the FocusDB system that ensures that all materialized view data is kept consistent with the collection data it is based upon. By doing so, the data fetching operations are able to return the most recent version of the data objects, keeping the user well informed.

The View Manager is an asynchronous component of the server-side module that is comprised of a single *Main Manager* thread as well as several *Worker* threads. Whenever

a new materialized view is instanced in the system, the Manager thread launches a new worker thread that will be responsible for executing the necessary operations to keep that view up-to-date. This strategy enables the splitting of the update load between several components, which improves the writing performance, since the number of concurrent aggregation and write commands issued to the MongoDB component is increased, allowing a better utilization of the capacity of the MongoDB client. At any given moment, there are as many worker threads running as there are views in the system.

4.5.1 Main View Manager

The view update process, exemplified in figure 4.1, starts with the FocusDB Data Layer, responsible for handling the logic behind the API read and write operations, triggering the view manager whenever a data object is inserted, updated or deleted from a collection. It does so by notifying the main Manager thread with a Collection Update object, which contains the name of the affected collection, the value of the object previous to the writing operation, i.e. its old value , and its value after the writing operation - the new object value. The type of writing operation is possible to be inferred from the old and new object values: an *Insert* will hold a new value and no old value, since the object did not exist before the operation; a *Delete* will hold only an old value, due to it no longer existing in the collection after the write, and finally an *Update* will hold both its old and new values.

The distinction between types of write operations matters in the context of the main View Manager, because its function is to translate a collection write operation in several view writes, in order to distribute them between its workers, and the types of collection writes will impact the view operations that need to be performed. The following list details the view operations that are required for each type of collection write:

- **Insert**:

  Starting with an example, suppose there is a system with a collection for parking space objects and two materialized views: one based on the attribute 'Street' with full object fidelity and one based on 'Neighbourhood with only a count of the available parking spaces. A new data object is inserted into the main collection, which has three attributes (besides its ID and location parameters): Street *A*, Neighbourhood *beta*, and Parking_Difficulty *medium*.

  The desired outcome is that both entries *A* and *beta* in the respective views be updated in order to reflect the existence of the new object. This would correspond to adding the object to the *A* entry and increasing the counter in entry *beta*. In order to be able to achieve this, the system must first understand what views need updating, so that it can command the View Manager workers to perform the modifications.

  The Manager starts by querying the view metadata collection for all views with a key attribute present in the inserted object. In this case, 'Street' and 'Neighbourhood' would return their respective views and 'Parking_Dificulty' would not. Then, it

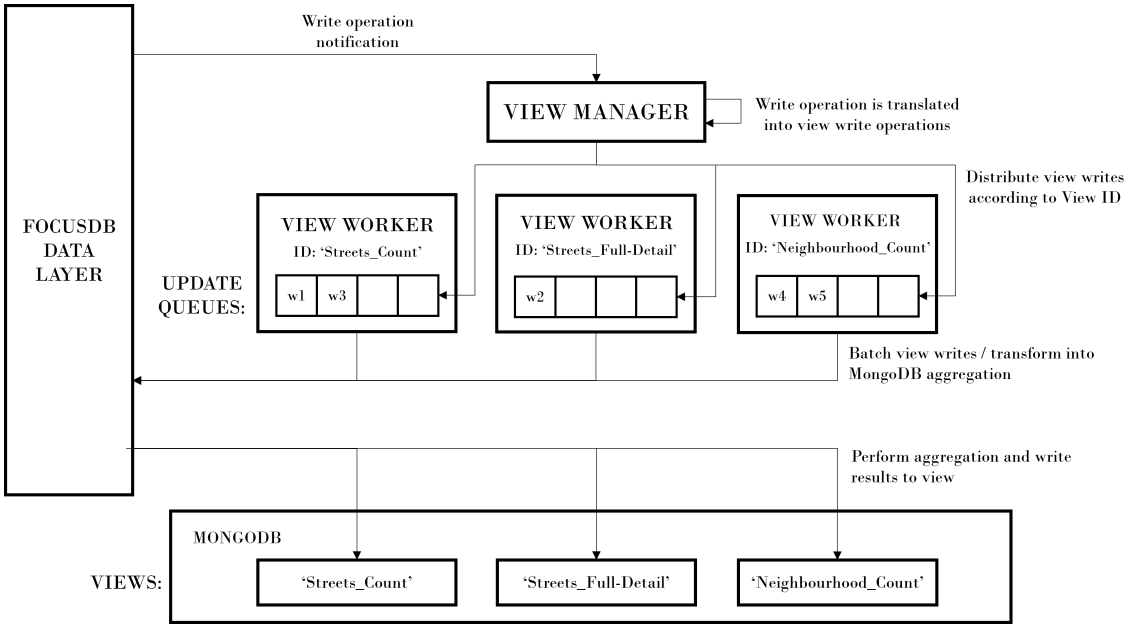

Figure 4.1: View Manager Work Flow.

signals the workers responsible for those two views, detailing which entry requires recalculating, by placing in their Update Queues the entry's key value. This concludes the responsibility of the main thread, leaving the modification of the view to be performed by the subsequent components.

- **Delete**:

  In the case of a Delete, the translation process is the exact same as with Insert. The removal of a data object means all views with entries containing the object in some form must be updated to no longer show its existence in the data. Therefore, the same steps are performed as in Insert: finding the views that need to be changed, and then placing the attribute value in the queue of the worker for each view, so that the respective entry no longer holds that data item or any lower fidelity version of it.

- **Update**:

  Finally, Updates are slightly more complex than the other two operation types, since they require dealing with attributes that act as key in a view and with the ones that do not. Suppose there is now an update on the object inserted previously, where the Neighbourhood is changed to *theta* and 'Parking_Difficulty' to *low*.

  This situation implies two distinct tasks. The first is to act on the views where a key attribute was modified, similarly to Insert and Delete. In this case, we know the object 'resides' in an entry already, so the course of action should be to recalculate that entry, as well as the one the object should belong in now. The second situation,

however, has to do with the level of fidelity of the views and its implications on the update process. For example, the Street view defined earlier is a view that holds an array of objects that expose the same level of detail as the Base Data objects. With this update, the 'A' entry will continue to incorrectly show an object where the Neighbourhood attribute is *beta* and 'Parking_Difficulty' *medium* because it is not a view where the key attribute was modified. To make sure all references of the object are updated, the system must also operate on all other views in the system, modifying the entries that hold the outdated objects. The system has knowledge of what entries to update, since their key attribute values correspond to the values of the attributes that were not modified in the updated object, which in this case is the 'A' entry of view Streets.

This process is still not completely optimized, since the View Manager has no way of ascertaining a view's fidelity level from its definition, since that is embedded in the MongoDB pipeline. As such, it acts with a greedy strategy, updating one entry in all views in the system, regardless of fidelity level. In the event a key attribute of a view is modified by the update the number of view updates is increased by one, to account for the old and new value entry updates. This compromise is deemed acceptable, mainly due to the fact that the View Manager workers batch operations on the same view in a very efficient manner. However, in a system with a large number of views, it would be sensible to extend the view definition to incorporate a representation of the level of fidelity of each view, so that only the necessary views should receive updates.

From this point on, the original collection write operations have been successfully translated into standard view operations, and those operations have been distributed to the correspondent View manager workers which are now responsible for continuing the update process.

4.5.2 View Manager Workers

The View Manager Workers are single threads that operate asynchronously to the Main View Manager, whose sole purpose is to ensure that the view they are assigned at startup remains consistent throughout the execution of FocusDB, or until said view is destroyed.

It possesses an Update Queue, a blocking queue that holds the values corresponding to the entries of the view that need to be recalculated. As soon as the worker is available, it polls the queue for a new batch of updates. If the queue is empty, it blocks the worker's request, holding until a new update is received, so that the worker does not needlessly expend CPU time, executing its endless loop. If it is not empty, it either returns a number of updates equal to the batch size, or returns all of its contents if it is holding less updates than the configured batch size. This batch size can be configured according to the needs of the developer and the cadence of writes received by the Data API.

After it receives the update batch, the worker then bundles them in a MongoDB pipeline. This pipeline starts as the aggregation pipeline set in the view definition, which dictates the level of fidelity of its data, but in order to only have an effect on the necessary entries, a *match* stage is added as the first operation in the sequence.

Finally, the worker sends the pipeline to the FocusDB Data Layer, which then orders the MongoDB client to perform the aggregation. The resulting data is then written to the corresponding materialized view, finalizing an iteration of the view management cycle. At this point, if the update queue is empty, the view is consistent with its main data collection and is ready to be queried for updated information.

4.6 Notification System

The final major component of FocusDB is the Notification System. Its purpose is to alert the FocusDB client when any update occurs on data that it is interested in, so that it can fetch the most recent version from the server. It communicates with the client through a Web Socket, using a purpose built protocol specific for this use case. Meanwhile on the backend, it employs a message broker named RabbitMQ [37] to queue the notification messages, which are then consumed by the notification system and sent to the client through the appropriate Web Socket channel.

4.6.1 Web Socket Protocol

The interaction between the client and the server-side components of the notification system starts with the client application deciding to perform a data request to the API. It uses the FocusDB client to fetch the data through the exposed client library, which contacts the API using its REST client. If the client application recognizes that it needs to keep that data updated, in the event of concurrent updates, it can manifest its interest in that data through the Start Data Notifications method of the client library. A call to this method begins the Web Socket interaction between client and server using the FocusDB Web Socket Protocol (to note that the Web Socket connection is established before this method is ever called, at the moment when FocusDB Client is first instanced). The protocol, detailed in figure 4.2, is composed of a simple set of messages, detailed as follows:

- **START** - A message to be sent by the client to the server. It indicates that the client is interested in establishing a new flow of notifications for a given data request, the latter which is attached to the message. To note that a client can initiate notification flows at any time and for any number of data requests. The set of requests with active notification flows is called the client's *Interest Set*.
- **START_SUCCESS** - After a START message is issued by the client, the server registers the new addition to that client's interest set in the notification system backend,

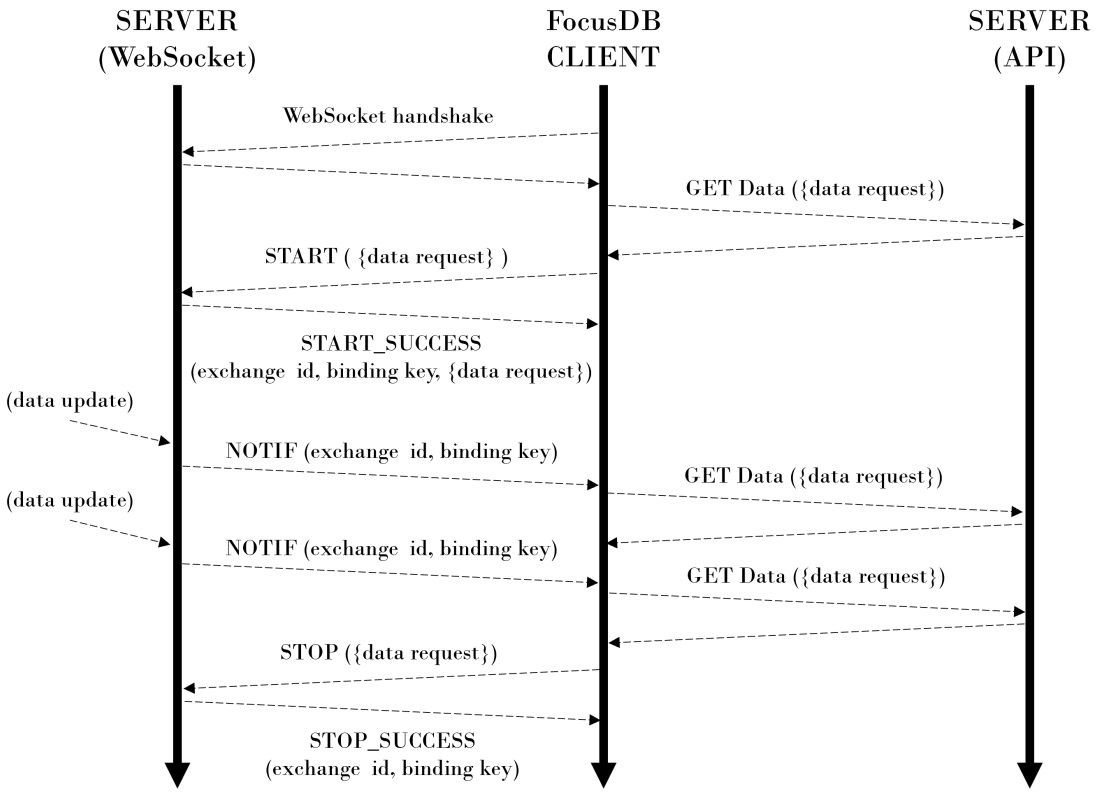

Figure 4.2: Notification System Protocol (detailed).

and issues the return of the START_SUCCESS message. Besides the initial data request, the server also attaches two other parameters to this message: the Exchange ID and the Binding Key.

The nomenclature is explained in the upcoming subsection 4.6.2, but succinctly, these two parameters act as a key for that data request, where Exchange ID is the ID of the view the request is targeting and Binding Key is the key of the view entry returned by that request. If a request returns more than one view entry, the server returns as many START_SUCCESS messages as view entries in the request. The client then maps the Exchange IDs and the Binding Keys with the data request to be used when notifications are received.

- **NOTIF** - This message is sent by the server to the client whenever a view update is triggered over the data that is of interest to the client; that is, data that corresponds to one of the requests in the client's interest set. Every NOTIF message carries an Exchange ID and a Binding Key, so that the client is able to ascertain which data request it must repeat, in order to receive the data corresponding to the notification. When this message is received, the FocusDB client automatically performs the corresponding request and updates the cache to reflect the newly updated information.

- **STOP** - When the end user is no longer interested in a certain set of data, it can order the FocusDB client to stop a flow of notifications associated with a data request. The client library which exposes the Stop Data Notifications method, sends a STOP message through the Web Socket, informing the Notification backend to cease the notification for that request.

- **STOP_SUCCESS** - Finally, after the backend stops a given notification flow, it returns a STOP_SUCCESS message that holds the exchange ID and the Binding Key of the of the ceased request. Once again, if the request corresponds to multiple binding keys, several STOP_SUCCESS messages are returned to the client.

4.6.2 Backend Implementation

As to the server-side module of the notification system, the main workflow centres around generating notification messages when view updates are conducted, utilizing the queuing capabilities of RabbitMQ to filter the notifications by the data request they are associated to, and finally, consuming the messages from the queues using callbacks and sending the notifications to the respective clients.

To aid in the process of notification distribution, we decided to employ a specialized message broker called RabbitMQ. The reason why RabbitMQ was chosen was because it had the capability of filtering the notifications according to a specific set of assigned values, through a technology called *Exchanges*. RabbitMQ is based on AMQP 0-9-1 [38], which is the messaging protocol that enables client applications to communicate with the middleware RabbitMQ brokers. It imposes that there must be components that feed messages to the broker, called *Producers*, as well as components that process those messages, called *Consumers*. Messages are published to *Exchanges*, which the authors relate to 'mailboxes', that distribute message copies to queues using rules called *Bindings*. The broker then either delivers messages to consumers subscribed to queues, or consumers fetch/pull messages from queues on demand.

In the context of FocusDB, an exchange is established for each view defined in the system, meaning an exchange is identified by the ID of each view. Several queues are then bound to the exchanges, utilizing a value taken by the key view attribute as the Binding Key. This allows the system to produce notifications, distribute them to the appropriate message queue, and finally have them be consumed and sent to the clients subscribed to that queue.

The process of subscribing a client to a queue, however, requires some processing. The goal is to be able to receive a data request and be able to translate it into the queues associated with the appropriate data entries in the materialized views, so that, when the entries receive an update, so does the client. The issue with the translation process is that it is not always straightforward.

As mentioned before, a data request can take one of two types: 'attribute' and 'location'. For type 'attribute' the process is rather undemanding; we simply take the query attribute

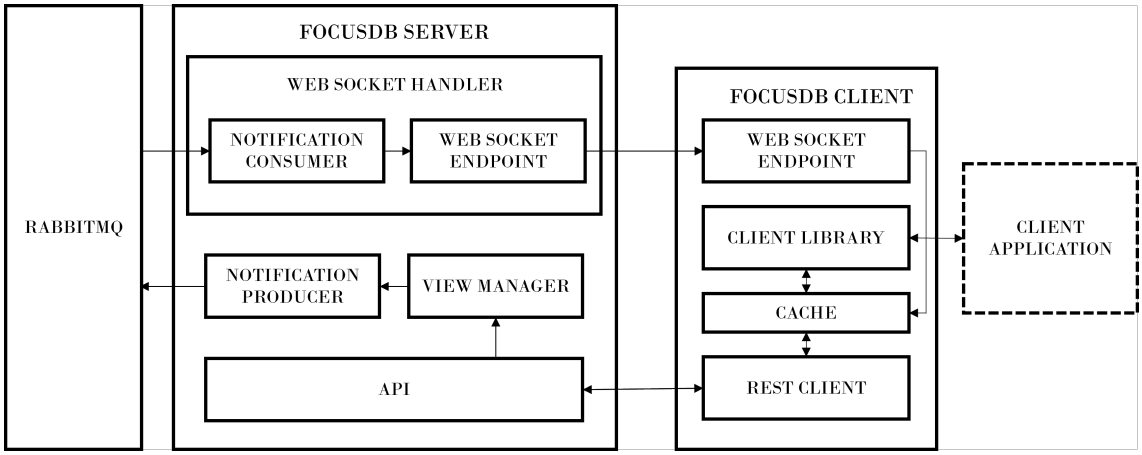

Figure 4.3: Notification System Client-Server Interaction Diagram.

and parse the entry ID that it contains. The challenge comes from requests with type 'location. In these cases, it is necessary to translate the geographic area into possibly several view entries, which the system does with the assistance of the centroids defined earlier. The system uses the geographic area to directly query the centroid collection with the help of the Control Data Layer, which results in the set of keys that correspond to that request's view entries. As such, it is possible to subscribe a client to any queue, given only the request for the data it is interested in.

Furthermore, the reason why the START_SUCCESS and STOP_SUCCESS protocol messages carry the Exchange ID and Binding Key is so that the client knows to associate the data request with those two parameters, so that when a notification is received, it is able to pick the correct data request to fetch the appropriate data.

In summary, the Notification System executes as detailed in figure 4.3. First, the API receives a call for a data write operation, which causes the View Manager to be triggered and the respective views to be updated. After a successful computation of a batch of updates, the View Manager workers must use the keys of the entries it just updated to start the flow of notifications. It contacts the Notification Producer and provides it with the modified view's identifier and key attribute values, which it then utilizes to place the newly created notifications in the corresponding message queues. RabbitMQ performs its duty by delivering the messages to the appropriate consumers, implemented by the Notification Consumer object. Whenever a new Web Socket connection is established, the system instances a new Notification Consumer and establishes a RabbitMQ callback on the appropriate queue. Once the queue is fed a notification object, the broker then delivers it to a consumer through the callback. The Consumer concludes the process by sending the notification through the Web Socket to the FocusDB Client, which then repeats the appropriate data request and updates its cache with the new data. The client application can request the data using through the Library, and the client will fetch it

from the cache, so long as it hasn't been removed or invalidated due to a timeout, in which case the request must inevitably be repeated.

4.7 FocusDB Client

Despite the FocusDB client being the only main component present on the client-side, it performs a critical role for FocusDB. Besides the previously explained WebSocket endpoint, it houses two other main components, the Data Cache and the Client Library.

The Data Cache allows the client to hold data in memory for an limited amount of time, which boosts performance by decreasing the load on the server module caused by repeated requests to the API. Most of the time, these requests originate from moments of user interaction with the mobile application, so it is essential that the system is able to hold information on the client side, in order to avoid moments where the user is kept waiting for data to be loaded.

On the other hand, without a method of holding onto data, the entire concept of notifying the client would not make sense if the intent is to improve performance, due to how often the client would have to repeat the data requests. Merely changing screens in an application would trigger the fetching operation, which is such a frequent action, that server-side updates to the data would quickly be reflected on the client-side, albeit while imposing a much greater load on both the network and the devices running the system.

The FocusDB data cache operates similarly to other caches: it is a component with limited space that holds onto data in memory for a limited amount of time, so that it can be accessed repeatedly by the client application without much resource cost. It follows a LRU (Least-recently-used) strategy when it reaches full capacity, choosing to free the object that was accessed least recently. It also evicts objects from memory, even if the cache is not full, if they remain too long without being accessed or updated. Its interaction with the notification system consists of preserving the result of an API data request and then invalidating it once a notification is received, substituting it with the updated results. All calls to the Get Data library method first check whether the cache already has the data in memory. If not, then it proceeds with the request to the API, placing the results in memory afterwards.

The Client Library aims to help with the transparency of the system, as well as with the usability of the data fetching mechanisms. If there were no FocusDB Client, application developers would not only have to manually integrate the Web Socket connection and its protocol, they would also need to handle the integration with the caching mechanisms and the calls to the Data API. Instead, with the client, apps need only utilize the methods exposed in the library, in order to take advantage of the benefits offered by FocusDB. The exposed methods are detailed below:

- **Get Data**: receives a data request with the same format as the Get Data API endpoint (as detailed in section 4.4.2). It first checks whether the cache already possesses

the results of that request in memory, and if it does not, it uses the REST client to contact the API to fetch the data.

- **Start Notifications**: Starts the flow of notifications through the sending of a START WebSocket protocol message, containing the data request that corresponds to the data in question.
- **Stop Notifications**: Stops an ongoing notification flow throughthe sending of a STOP WebSocket protocol message, containing the data request that corresponds to the data in question.
- **Close Session**: Stops all notification flows and closes the WebSocket session with the server side endpoint

This concludes the Implementation chapter of this dissertation. The following chapter concerns the Evaluation phase of the thesis work, and will provide an exposition of the system's performance results obtained during the experiments, as well as describe the methodology followed and the use-case scenario considered.

5

Evaluation

This chapter presents the evaluation work carried out following the implementation phase. In it, we will explain the thought process behind the structure of the tests, as well as the methodology and technologies used to evaluate the system. Finally, the performance metrics obtained from the testing of FocusDB will be compared to ones gathered from industry standard solutions, in order to determine whether the methods proposed in this dissertation yield any performance gains.

5.1 Use Case Application

FocusDB's unique approach makes it suitable for various application scenarios. For instance, FocusDB can provide real-time data sharing based on the location of users in multiplayer interactive games [32, 22], where players can subscribe to data near their location, and the system can adapt the consistency and exposed detail of the data based on locations to ensure optimal performance. FocusDB can also aid mapping services [13] by filtering and delivering relevant information to users by implementing real-time crowd-sourced traffic and alert information filtered based on the current location of the used. Finally, it also able to adapt the level of detail of the data based on device location in hyperlocal data distribution systems [9, 52], saving bandwidth for the client and costs for advertisers.

For the purposes of testing the system, we imagined a location-based application that showcases the need for a new consistency model as the use case. The next subsection describes this application in detail.

5.1.1 Where to Park?

For many drivers in densely populated areas, finding a parking spot is a daily challenge. Whether commuting to work or visiting an unfamiliar place, knowing where to park can make all the difference. That is the motivation for our envisioned use case, the *Where to Park?* application.

The Where to Park? application is an imagined mobile application for smartphones and automobiles that keeps track of and helps find empty parking spots. The app provides users with multiple levels of detail on parking availability across in different regions, from city-wide to street-level views. Users can see an approximate number of available spots for an area or zoom in on the street and see all the parking locations and their availability. The app can be used for both street parking (free or paid) and private parking facilities (e.g. shopping malls). For private parking, the app can retrieve real-time data on availability through an API provided by the parking operator. For street parking, the app can rely on a network of sensors installed in each spot that communicates with nearby parking meters or crowd-sourced data from users who report available spots.

5.2 Methodology

The experimental portion of this work was conducted as an evaluation of the performance of FocusDB in a mobile setting, based on the previous use-case scenario. For this, we intended to test the system with two separate goals. First, we wished to compare the performance of the FocusDB data model, that is based on the fetching of data from precomputed materialized views, with the standard approach of fetching the data directly from a database collection. After that, we also wished to ascertain whether the notification system would provide any performance benefits over a simple repetition of the client data requests.

5.2.1 Experimental setup

FocusDB was equipped with the Incremental View Manager and also backed by a MongoDB database, where the parking spot dataset was seeded. In order to evaluate the system in a manner that is consistent with real-world patterns, we also built a workload generator capable of mimicking multiple mobile clients simultaneously. This workload generator can launch a configurable number of clients, where each runs a predefined set of trips and workloads, providing a realistic test to FocusDB's capabilities. To enhance the test environment and introduce an element of unpredictability, we implemented a server-side Ghost Client mechanism (previously mentioned in subsection 4.4.4). This mechanism randomly chooses a number of parking spaces from the database and inverts their availability, keeping the dataset dynamic and invalidating cached data. The number of affected objects and the time interval between updates are configurable parameters, which were set to 100 parking availability changes every two seconds during experiments.

5.2.2 Geographic datasets

As to the data, the goal was to be able to provide a set of data objects that allowed the workload generator to create realistic client movements and data object modifications. As such, we chose to leverage two real datasets. The first includes the geographical

coordinates of roughly 500 taxi trips in the city of Porto, Portugal, and was subsequently enriched with the assistance of the OpenRouteService API [46] to reflect more detailed geographical information, such as the names of neighbourhoods and streets. The second is derived from the initial taxi trips dataset, and contains the coordinates of approximately 500,000 parking locations in Porto. This ensures that the simulated vehicles performing the itineraries described in the first dataset are able to intersect parking spots. The taxi trips dataset is, therefore, used to compute client movements and the parking spots dataset to represent the locations of relevant data objects in the database.

5.2.3 Technical details

All experiments were carried out using the Nova SST Computer Science Department's Compute Cluster, which offers a variety of machine types. Both the server-side and clients were deployed on machines equipped with an AMD EPYC 7281 CPU (16 cores/32 threads), 128 GiB DDR4 2666 MHz of RAM, and dual 10 Gbps network capacity.

The experiments are conducted using one server machine and one or more client machines. The server is deployed using a docker-compose configuration, which launches FocusDB, MongoDB and RabbitMQ, each in its own container. FocusDB communicates with the MongoDB and RabbitMQ containers through their respective default ports, 27017 and 5672, and exposes port 8080 to enable clients running on different machines to have access to the API. The client machines, the number of which varies depending on the workload, run a single instance of the workload generator responsible for managing the lifecycle of the multiple simulated mobile clients. During the workload, Each simulated client makes requests to the server derived from the taxi trips dataset. Each of the testing workloads are detailed in section 5.2.4.

Running an experimental workload requires a workload configuration file, which details the machines to run the server and client modules, the workload specification to run, and other relevant parameters, such as the dataset file path or the speed of the simulated trip.

The experimental process is carried out using a python script that deploys the server and client modules, and commands the workload generators running in the clients to start the specified workloads. This script takes three parameters, the workload IDs, the number of clients to be simulated in each client, and the mode in which to run the server (the notion of server mode is also workload dependent and is explained in the next section). The script launches the clients with the workload configuration corresponding to the workload ID and then waits for all clients to finish their simulations. Since the machines in the cluster share a file system, the clients need only create a small text file in a predetermined directory to inform the server that their execution has finished. The files are then deleted after all client machines finish, to allow for subsequent runs.

Before each run of a workload, the MongoDB database needs to be reset. This is done through the use of the methods detailed in section 4.4.3. The MongoDB database is seeded

with 500,000 parking spots, 70% of which are available, and views are precomputed at the neighbourhood and street levels. After the reset process is complete, the client can start the experiment. Clients communicate with the API through URLs composed by the cluster machine ID, followed by the 8080 port, and then the path to the corresponding API resource. During the workload, the client gathers performance metrics of the system, such as latency and throughput and averages them over the course of the workload, which it prints to a CSV file, once the workload is complete. The result CSV files are then aggregated and compared against data from other workload configurations.

5.2.4 Workloads

As mentioned in the start of this section, the testing of the system is split between two main questions:

- Is the FocusDB data model more efficient at providing data to mobile clients than the standard models?
- Is using a notification system a more efficient way to propagate data, in terms or resource usage, than a periodic repetition of data requests?

In order to answer these questions, we developed a series of workloads to evaluate the system's performance in each scenario. We start by describing the workloads that regard the data model, followed by the notification system workloads. All of the workloads described below concern the performance of reading operations. Although writes are performed during the experiments, the main focus is to evaluate reads, since that is the main purpose of the FocusDB implementation.

5.2.5 Data Model Workloads

For the evaluation of this component, two additional data models were developed, serving as a representation of the current industry standards. Instead of utilizing materialized views, the mode dubbed as BASIC queries the parking space collection directly. Similarly, INDEX mode also queries the data collection but takes advantage of indexes created over the main data attributes, 'street' and 'neighbourhood'. Both of these return only full representations of the data objects, and do not employ any sort of fidelity reduction. For the purposes of comparing the three alternatives, the FocusDB data model is labelled as FOCUS in the plots.

Two workloads were developed for this experiment. Both simulate a user moving from one location to another, which correspond to the path described by a taxi trip from the aforementioned dataset. While the client is moving, it is performing as many data requests to the API as it can handle, with the intent of receiving the parking spot situation in the destination location.

Workload 1: The goal of this workload is to test whether reducing the level of detail of the data received can influence performance positively. As such, for mode FOCUS, for the preceding 80% of the trip distance, the client requests data from a view with low data fidelity, where each entry represents the number of parking spaces in that area. For the last 20%, the client changes the requests to a view with the complete data object representation. This goes in line with the expected user behaviour, where, while the user is far away from its destination, it is not interested in receiving the exact parking spots details. As it moves closer, those details, such as the location of the spaces, start becoming more important, so the workload mimics the application changing the level of detail to fit the needs of the user. For modes BASIC and INDEX, the requests always generate responses with complete data detail.

Workload 2: In the second workload, the goal is to determine whether exclusively performing requests with complete detail is detrimental to the performance of the FocusDB data model. By not adapting the fidelity level, all three modes, FOCUS, BASIC and INDEX are tested at equal footing, meaning the amount of fetched data stops being a factor for this experience. We are intentionally evaluating the performance of FocusDB in a manner that is antithetical to is design motivation, in order to ascertain whether it disadvantageous to use our implementation over the standards in situations more in line with a regular pattern of access. As such, this workload consists of a client following a trip from start to end, while performing data requests to the destination location with full object fidelity. In mode FOCUS, all requests are made to the high fidelity view, while in the other two modes, the requests remain as they did for Workload 1.

Both workload 1 and 2 only require the usage of one client machine running the workload generator, since by scaling the number of client threads we can easily achieve server saturation on either of the server implementations. The client performs 5 trip simulations and then averages the results between them, in order to remove outliers.

5.2.6 Notification Workloads

The evaluation of the notification system's effects on performance requires two server modes, one where the Notification system is active and one where it is not. In the notification-enabled mode, named NOTIF, the system takes advantage of the notification system and the client cache, in order not to send repeated data requests. Instead of repeating the request, the client fetches the data stored in its cache, which is kept updated by notification flow with the server. The other mode, NO-NOTIF, simply repeats the data request to the API, since it cannot store the data responses nor receive notifications. Both of the modes utilize the FocusDB Client to fetch data, utilizing the exposed library instead of requesting the API directly.

Two more workloads were designed for this experiment. They follow the same structure as before: a client simulates a trip from the taxi dataset, where the client fetches

data as it is moving. To note that, in mode NOTIF, a repeated data fetch order usually corresponds to a cache read and not an API request. Because of this, and in order to make the comparison fairer between the two modes, the number of data fetch operations per position of the trip is now limited to 5. A user performing more than 5 data fetches per position would not be realistic in the context of our use case, and doing so would have uneven consequences between the two server modes, due to only one having a cache.

Workload 3: This scenario is very similar to the one in Workload 1. The client fetches parking space data for the destination of the trip, with the first 80% corresponding to low detail level requests and the last 20% to high detail level requests. The goal is to expand on the results of Workload 1, by pondering what the performance of the system is, now that the load of the server module has theoretically been decreased, when compared with the base version.

Workload 4: The final workload also adheres to the previous experimental format, by having the client requesting the destination location with varying levels of data detail, as it is moving. However, this workload also forces the client to fetch the parking space data around its current position, with there being an equal chance of the client querying for its current position or for the destination. To note that, since the user is constantly moving, it does not make sense in the context of the use-case application, for the client to receive notifications for the requests it makes to its current position.

The purpose of this workload is to measure the evolution of the performance metrics when increasing the load on the server, by performing extra mid-trip requests. We expect a dip in performance, relative to the results in workload 3. However, the main objective is comparing how the two modes as the number of clients increases.

This following section details all results obtained during the evaluation phase.

5.3 Results

This final section contains the results obtained from the testing of the FocusDB system according to the previously listed workloads. The graphs presented herein expose the performance metrics obtained from in each test, in each server mode, and will be used to determine whether FocusDB demonstrates an improvement over the standard solutions.

5.3.1 Data Model Evaluation Results

We begin with the results obtained from the testing of the data model through the API. The experiments measure throughput and latency, evaluated as a function of the number of clients, and hence, the total number of client requests.

5.3.1.1 Workload 1

Starting with Workload 1, figure 5.1 presents a throughput × latency plots comparing all alternatives, showing that using FocusDB (FOCUS) leads to improved performance when compared with the BASIC and INDEX alternatives. The plot shows that the mechanisms employed in the design of FocusDB allow to support a similar number of clients, while offering lower latency and achieving a higher throughput. The same can be said when analysing graphs 5.2 and 5.3, where for the same number of clients (nodes), FocusDB displays a lower latency and a higher throughput value. The BASIC alternative exhibits the lowest throughput and highest latency among the three alternatives, whereas the INDEX alternative, although closer to FocusDB, still displays a very significant gap in performance.

This gap can be explained by the plots in figures 5.4 and 5.5, which capture the number of requests per number of clients and the response size per number of requests, respectively. Since FocusDB is able to adapt the level of detail of the requested data objects, as is shown in plot 5.4, the size of the responses it is able to produce is significantly smaller to the two other alternatives. This, in turn, leads to a more effective usage of network resources, as well as the computational resources that return those answers, which then translates into the server being able to service more clients in the same time span, while also providing quicker responses.

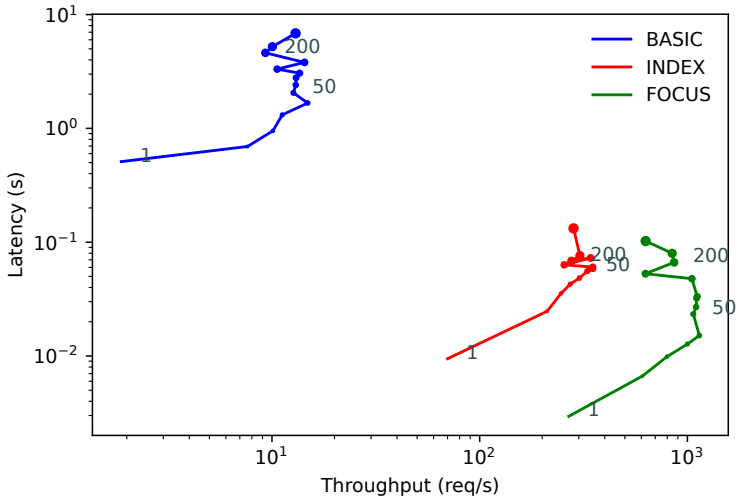

Figure 5.1: Throughput Latency per number of clients - Workload 1

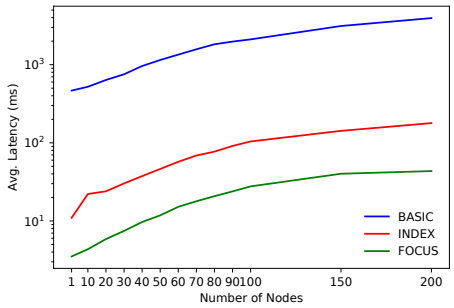

Figure 5.2: Latency per number of clients - Workload 1

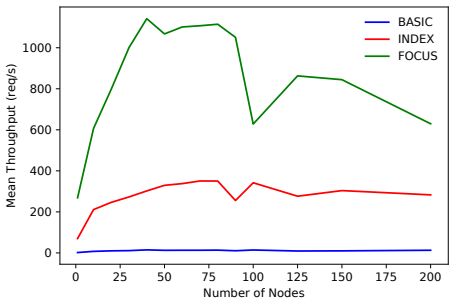

Figure 5.3: Throughput per number of clients - Workload 1

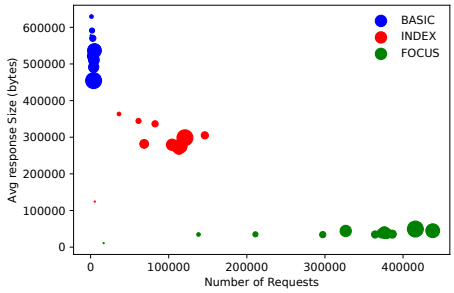

Figure 5.4: Response size per number of requests - Workload 1

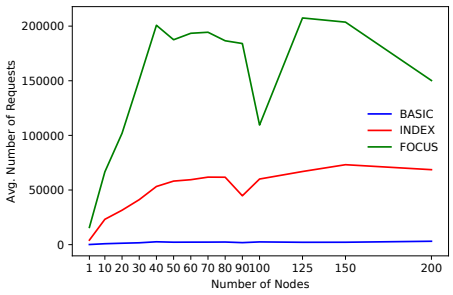

Figure 5.5: Number of requests per number of clients - Workload 1

5.3.1.2 Workload 2

As to Workload 2, although not as pronounced as in the first example, the FocusDB implementation still manages to exhibit better performance metrics that the other two alternatives. When it comes to latency, as shown in figure 5.8, FOCUS and INDEX demonstrate a very similar pattern, with BASIC being significantly worse. However, when we move over to throughput (5.7), the gap between the two top alternatives widens, showing that the server is struggling to perform the query operations in INDEX mode at a similar rate to FOCUS. Hence, the throughput x latency graph (5.6) shows the plots for the FOCUS and INDEX modes being closer to each other, while still exhibiting a small performance gap, with BASIC displaying an almost incomparable performance to the other two, which is to be expected since MongoDB has to query the entire database of 500,000 entries every time a data request is made, instead of using materialized views or indexes.

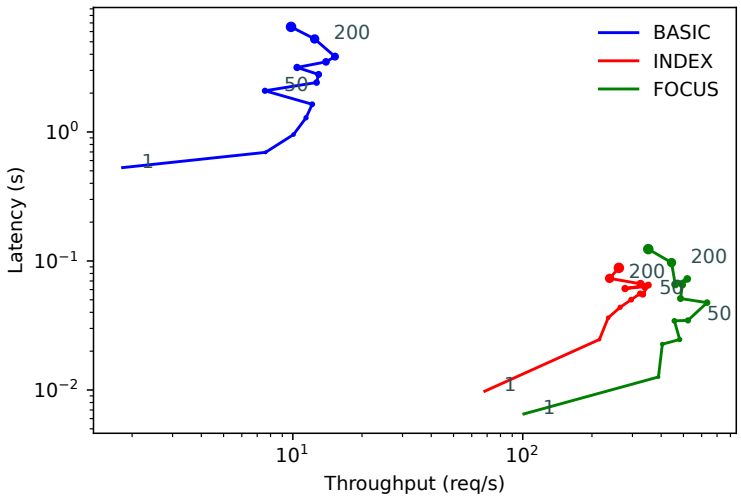

Figure 5.6: Throughput Latency per number of clients - Workload 2

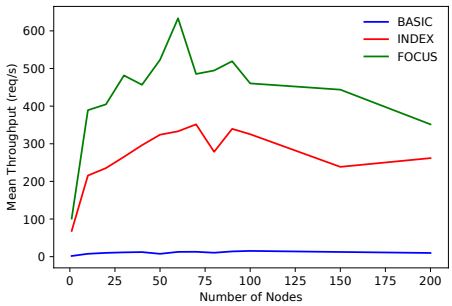

Figure 5.7: Throughput per number of clients - Workload 2

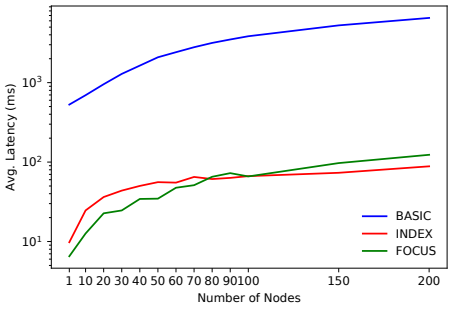

Figure 5.8: Latency per number of clients - Workload 2

5.3.2 Notification System Evaluation Results

As to the notification system workloads, the results are less clear cut than in the previous cases.

5.3.2.1 Workload 3

We can see in figure 5.10 that in mode NOTIF, the latency starts much lower than in the mode with no notifications. This is because the great majority of requests will result in cache hits in mode NOTIF, while in the opposite case, all requests are made exclusively to the data API, resulting in a higher latency as expected. However, as the number of clients surpasses the 150 mark, the performance of the notification mode quickly degrades. Similarly, in graphs 5.11 and 5.9, up until the 150 client mark, the NOTIF mode throughput values match the NO-NOTIF ones, since the server has not yet been saturated. We expected the NO-NOTIF mode's throughput to invert its tendency first, since it is inherently placing a larger load on the server and the network. However, the

opposite occurs, where the throughput values for mode NOTIF stop increasing after 150 clients.

The reason behind this unexpected worsening of the metrics can be attributed to a problem in either the Web Socket system, the Notification Backend or the cache. This assessment is justified by the fact that the performance in mode NO-NOTIF, which does not have access to the aforementioned components, does not appear to degrade at the rate exhibited by mode NOTIF. This problem has the consequence of the server not being able to scale to a number of clients that would cause the metrics of the notification mode to surpass those of the more naive alternative, which nullifies the theoretical benefits of the notification system.

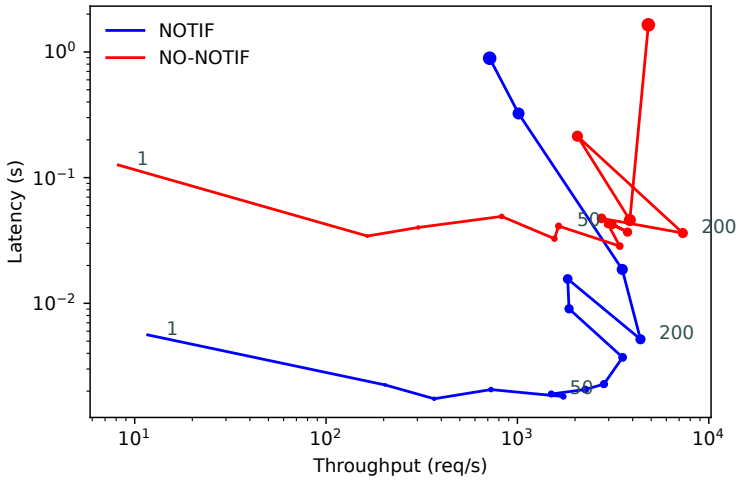

Figure 5.9: Throughput Latency per number of clients - Workload 3

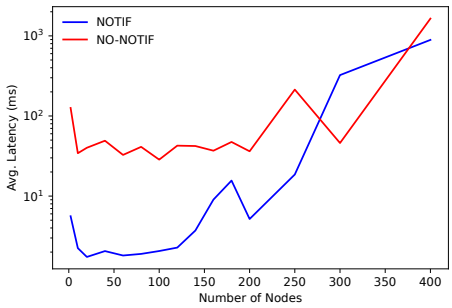

Figure 5.10: Latency per number of clients - Workload 3

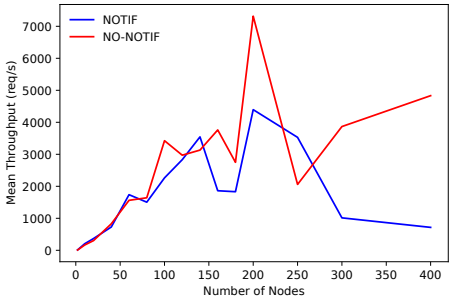

Figure 5.11: Throughput per number of clients - Workload 3

5.3.2.2 Workload 4

Similarly, in workload 4 the problems reappear, being somewhat exacerbated due to the extra server load caused by the mid-trip data requests. As shown in figures 5.12, 5.13 and 5.14, the latency in notification mode follows the same tendency of starting lower than in

NO-NOTIF, but then quickly escalates at around 75 clients. The throughput also follows the same trajectory, but in this case we are able to achieve saturation of the server in the NO-NOTIF mode.

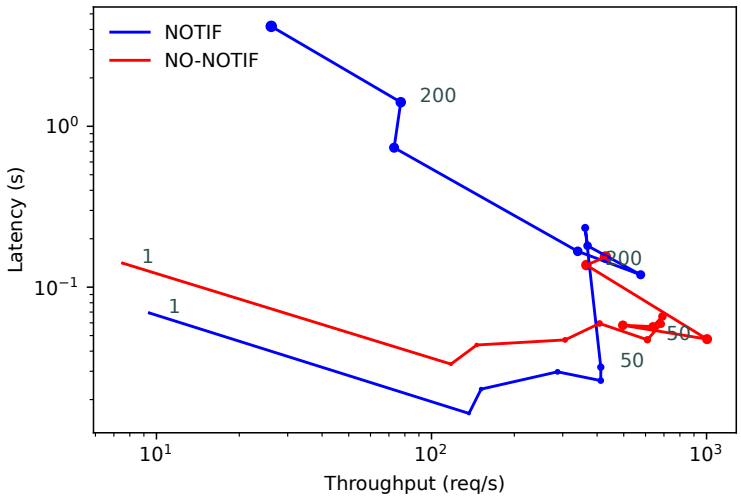

Figure 5.12: Throughput Latency per number of clients - Workload 4

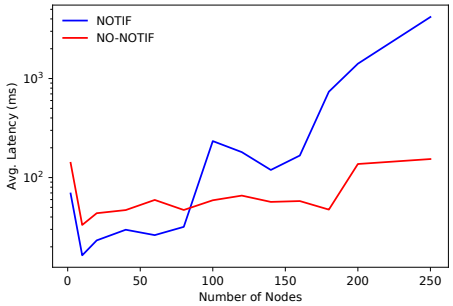

Figure 5.13: Latency per number of clients - Workload 4

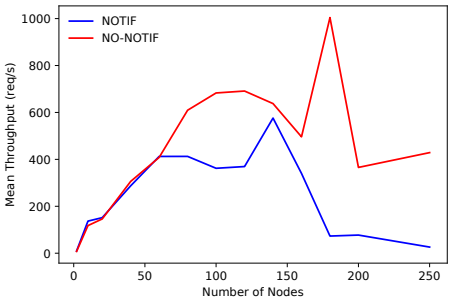

Figure 5.14: Throughput per number of clients - Workload 4

5.4 Final remarks

Unfortunately, due to this unexpected issue, the results for testing of the notification system do not correspond to our expectations. Although the beginning of the workloads exhibits promising data, the quick degradation of performance makes it so no accurate comparison can be made between the two approaches, leaving this experience inconclusive.

On the other hand, the FocusDB data model demonstrated very promising results, where the standard industry methods were sometimes much worse at dealing with the propagation of data to mobile clients than the solution proposed in this dissertation.

The final chapter will serve as a conclusion to this document, summarizing the system and the development efforts in the context of the results obtained in this evaluation phase.

6

Conclusion

This thesis work focused on addressing the ever-growing need of applications with location awareness for efficient data access and manipulation. In line with this necessity, we proposed FocusDB, a novel data management system specifically designed to support geo-located data in a mobile environment. Its powerful data model is capable of organizing data in a way that facilitates access by mobile users, and provides them with a adaptive notion of data fidelity, which aids in lessening the load on the server modules and associated networks. When allied with client caching mechanisms and a notification system, the system is able to adapt data delivery based on the location associated with data objects and the location of the user, as well as keeping application data up-to-date in a manner that ensures efficient access by the clients.

The experimental evaluation shows that FocusDB can achieve a higher performance than classical alternatives, through the leveraging of its data model, which allows to expose different levels of detail based on location. At the notification level however, the results obtained were inconclusive, due to some problems when scaling the implementation to a larger number of clients.

6.1 Future Work

This dissertation was the first step in a continued research effort that aims to expand the notion of location-governed distributed systems. The next step is to improve the performance issues surrounding the notification system, so that a conclusion can be reached on whether notification flows improve the resource consumption caused by repeated data requests.

After that, development endeavors could be focused on formalising consistency models that take into account the location of both data objects and servers/clients, as well as efficient replication protocols that can implement such consistency models. It may someday even be possible to apply such a system to a edge server infrastructure, bringing the relevant data closer to the clients that need it, or even to *Internet-of-things* applications, as an evolution of the interconnected world we live in.

Bibliography

[1] *Akka Serverless documentation*. https://developer.lightbend.com/docs/akka-serverless/index.html (cit. on p. 23).

[2] M. Armbrust et al. "A view of cloud computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58 (cit. on p. 8).

[3] S. Babu. "Continuous Query". In: *Encyclopedia of Database Systems*. Ed. by L. LIU and M. T. ÖZSU. Boston, MA: Springer US, 2009, pp. 492–493. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_85. URL: https://doi.org/10.1007/978-0-387-39940-9_85 (cit. on p. 31).

[4] *Cloud Firebase documentation*. https://firebase.google.com/docs/firestore (cit. on p. 24).

[5] J. C. Corbett et al. "Spanner: Google's globally distributed database". In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), pp. 1–22 (cit. on p. 33).

[6] *CouchDB documentation*. https://docs.couchdb.org/en/stable/intro/index.html (cit. on p. 22).

[7] A. Demers et al. "Epidemic algorithms for replicated database maintenance". In: *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing* (1987), pp. 1–12 (cit. on p. 11).

[8] *Ditto.live documentation*. https://docs.ditto.live/?_gl=1*129j3oc*_ga*MTE3NDYzNjc5OS4xNjM5MzA0NDEx*_ga_D8PMW3CCL2*MTY0NTExOTg1Ny43LjAuMTY0NTExOTg1Ny42MA.. (cit. on p. 21).

[9] *Foursquare. 2023. Foursquare City Guide*. https://foursquare.com/city-guide (cit. on p. 55).

[10] *GeoJSON Specification (RFC 7946)*. https://www.rfc-editor.org/rfc/rfc7946 (cit. on p. 36).

[11] T. K. George Coulouris Jean Dollimore and G. Blair. "Distributed Systems – Concepts and Design". In: 5th edition. Addison-Wesley, 2011. Chap. 2, pp. 37–79. ISBN: 0132143011 (cit. on p. 4).

[12] 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 (cit. on p. 10).

[13] *Google Inc. 2023. Nearby API.* https://developers.google.com/nearby (cit. on p. 55).

[14] *Graphite DB.* http://graphiteapp.org/. 2023 (cit. on p. 25).

[15] *HarperDB documentation*. https://harperdb.io/docs/overview/ (cit. on p. 22).

[16] M. P. Herlihy and J. M. Wing. "Linearizability: a correctness condition for concurrent objects". In: *ACM Transactions on Programming Languages and Systems* 12.3 (1990), pp. 463–492 (cit. on p. 11).

[17] *InfluxData. InfluxDB Times Series Data Platform | InfluxData.* https://www.influxdata.com/. 2023 (cit. on p. 25).

[18] *JAX-RS Oracle Documentation*. https://docs.oracle.com/javaee/7/tutorial/jaxrs002.htm (cit. on p. 38).

[19] A. Lakshman and P. Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS operating systems review* 44.2 (2010), pp. 35–40 (cit. on p. 2).

[20] L. Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Transactions on Computers* C-28.9 (1979), pp. 690–691 (cit. on p. 10).

[21] L. Lamport. "Time, Clocks and the Ordering of Events in a Distributed System". In: *Communications of the ACM* 21.7 (1978), pp. 558–565 (cit. on p. 12).

[22] J. Leitao et al. "Towards enabling novel edge-enabled applications". In: *arXiv preprint arXiv:1805.06989* (2018) (cit. on p. 55).

[23] *Lime, Electric scooter sharing app*. https://www.li.me/ (cit. on p. 1).

[24] A. van der Linde, J. Leitão, and N. Preguiça. "Practical client-side replication: weak consistency semantics for insecure settings". In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 2590–2605 (cit. on p. 20).

[25] A. van der Linde et al. "Legion: Enriching internet services with peer-to-peer interactions". In: *Proceedings of the 26th International Conference on World Wide Web*. 2017, pp. 283–292 (cit. on p. 19).

[26] *Macrometa documentation*. https://macrometa.com/docs/ (cit. on p. 23).

[27] *MongoDB Aggregation Pipeline*. https://www.mongodb.com/basics/aggregation-pipeline (cit. on p. 36).

[28] *MongoDB Aggregation Pipeline Operators*. https://www.mongodb.com/docs/manual/reference/operator/aggregation/ (cit. on p. 36).

[29] *MongoDB documentation*. https://www.mongodb.com/docs/ (cit. on p. 35).

[30] *MongoDB geographical queries*. https://www.mongodb.com/docs/manual/geospatial-queries/ (cit. on p. 36).

[31] *MongoDB Geospatial Indexes*. https://www.mongodb.com/docs/manual/geospatial-queries/#std-label-geo-2dsphere (cit. on p. 37).

[32] *Niantic Inc. 2023. Pokémon GO*. https://pokemongolive.com/ (cit. on pp. 1, 55).

[33] P. E. O'Neil. "The escrow transactional method". In: *ACM Transactions on Database Systems (TODS)* 11.4 (1986), pp. 405–430 (cit. on pp. 13, 14).

[34] *Parkify, Parking space locating app*. http://www.parkify.co.uk/ (cit. on p. 1).

[35] D. Perkins et al. "Simba: Tunable end-to-end data consistency for mobile apps". In: *Proceedings of the Tenth European Conference on Computer Systems*. 2015, pp. 1–16 (cit. on p. 18).

[36] N. Preguiça. "Conflict-free replicated data types: An overview". In: *arXiv preprint arXiv:1806.10254* (2018) (cit. on p. 12).

[37] *RabbitMQ Documentation*. https://www.rabbitmq.com/documentation.html (cit. on p. 49).

[38] *RabbitMQ Queuing Protocol*. https://www.rabbitmq.com/tutorials/amqp-concepts.html (cit. on p. 51).

[39] B. Rabenstein and J. Volz. "Prometheus: A {Next-Generation} Monitoring System (Talk)". In: (2015) (cit. on p. 25).

[40] K. Ramamritham and C. Pu. "A formal characterization of epsilon serializability". In: *IEEE Transactions on Knowledge and Data Engineering* 7.6 (1995), pp. 997–1007 (cit. on p. 14).

[41] R. Rodrigues and P. Druschel. "Peer-to-peer systems". In: *Communications of the ACM* 53.10 (2010), pp. 72–82 (cit. on p. 5).

[42] N. Santos, L. Veiga, and P. Ferreira. "Vector-field consistency for ad-hoc gaming". In: *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware* (2007), pp. 80–100 (cit. on pp. 15, 33).

[43] N. Santos et al. "FocusDB: Gestão de dados para aplicações móveis dependentes da localização". In: *Comunicações do 13º Simposio de Informática - INForum 2022*. 2022, pp. 104–114 (cit. on p. 3).

[44] N. M. Santos et al. "Data Management for mobile applications dependent on geo-located data". In: *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data*. 2023, pp. 70–76 (cit. on p. 3).

[45] M. Satyanarayanan. "The Emergence of Edge Computing". In: *Computer* 50.1 (2017), pp. 30–39 (cit. on p. 8).

[46] A. Schilling et al. "Integrating terrain surface and street network for 3D routing". In: *3D Geo-Information Sciences* (2009), pp. 109–126 (cit. on p. 57).

[47] W. Schultz, T. Avitabile, and A. Cabral. "Tunable consistency in mongodb". In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 2071–2081 (cit. on p. 2).

[48] D. Shukla et al. "Schema-agnostic indexing with Azure DocumentDB". In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1668–1679 (cit. on p. 2).

[49] L. M. Silva et al. "Geo-located data for better dynamic replication". In: *arXiv preprint arXiv:2205.01045* (2022) (cit. on p. 2).

[50] *SkySpark documentation*. https://www.skyfoundry.com/file/8/SkySpark-Overview-Brochure.pdf (cit. on p. 23).

[51] R. Steinmetz. "Peer-to-Peer Systems and Applications". In: Springer Science Business Media, 2005. Chap. 21, pp. 353–365. ISBN: 354029192X (cit. on p. 7).

[52] *Telegram Team. 2019. Location-Based Chats, Adding Contacts Without Phone Numbers and More.* https://telegram.org/blog/contacts-local-groups (cit. on p. 55).

[53] D. B. Terry et al. "Managing update conflicts in Bayou, a weakly connected replicated storage system". In: *ACM SIGOPS Operating Systems Review* 29.5 (1995), pp. 172–182 (cit. on p. 16).

[54] *Timescale Inc. TimescaleDB : SQL made scalable for time-series data.* https://www.timescale.com/. 2023 (cit. on p. 25).

[55] K. Veeraraghavan et al. "Fidelity-aware replication for mobile devices". In: *Proceedings of the 7th international conference on Mobile systems, applications, and services*. 2009, pp. 83–94 (cit. on p. 17).

[56] W. Vogels. "Eventually Consistent". In: *Communications of the ACM* 52.1 (2009), pp. 40–44 (cit. on p. 11).

[57] *Waze: Driving directions, live traffic road conditions updates.* https://www.waze.com/waze (cit. on p. 1).

[58] H. Yu. "Design and evaluation of a continuous consistency model for replicated services". In: *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*. 2000 (cit. on pp. 15, 33).