ECOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

MASTER'S THESIS

A robust and efficient data synchronization channel for mobile platforms

Author: Hadrien MILANO

Supervisors: Benoît BRIOT (ELCA[1])

Dr. Willy ZWAENEPOEL (LABOS[2])

March 15, 2018

[1]ELCA Informatique SA
[2]Operating Systems Laboratory, EPFL

Abstract

Nowadays, mobile network connectivity remains unreliable due to incomplete coverage, overloading, and other practical issues. Therefore, to be 100% available, enterprise mobile applications need to be usable offline. In this thesis, we analyze why this old problem is still awaiting a better solution and attempt to provide one.
We establish a set of requirements for a typical enterprise mobile application and use it to evaluate different architectures. We draw the blueprint for a solution which has the potential to overcome all the issues encountered so far. We analyze the state of the art and relate it to the architectures we have studied. We conclude that, while all architectures studied here have flaws, there is hope for future work to leverage recent research to bring a truly satisfactory solution to the problem.

Acknowledgements

I would like to thank both my project supervisors: Benoît Briot at ELCA who empowered me to conduct my work thanks to his trust and support, and Dr. Willy Zwaenepoel at EPFL who promptly answered my questions, ensured that I stayed on track, and showed interest when discussing design ideas.

Thanks to Kristina Spirovska for taking the time to explain her work on causally consistent databases and how it relates to the subject of my thesis.

Finally, I am grateful to Christophe Tafani-Dereeper and David Tang for proofreading my thesis and providing critical feedback.

Contents

Chapter 1

Introduction

Enterprise mobile applications have been around ever since the first portable digital assistants existed. Also, since the beginning until today, *offline capability* has been a critical objective, ans it will likely remain one for decades. Then why is writing offline-capable applications still a significant hassle after all these years?

In this thesis, we explore the design space surrounding offline-capable enterprise mobile applications down to the theoretical foundations. We review the state of the art and attempt to come up with a better approach improving developer experience. In doing so, we attempt to answer the question above and figure out why past attempts failed to bring the improvements we are seeking.

1.1 A revolution in enterprise mobile applications

Smartphones are nowadays a commodity possessed by most adult individuals of developed countries. This universal adoption led to a dramatic explosion of the price to performance ratio of such devices thanks to the economy of scale. While enterprise mobile applications used to run on specialized devices, the cost-effectiveness of smartphones led IT consultancy firms to use them for most of their mobile solutions. Being largely dominated by two operating systems, the technology brings together millions of developers[1] on a relatively small platform.

This concentration creates an unprecedented breeding ground for innovative technologies and open source initiatives. The potential for code reuse is vast, and a useful library can quickly be used by thousands of companies worldwide.

Despite the massive hardware improvements of the past years, network connectivity remains unsatisfactory for many use cases. Enterprise mobile applications, in particular, are susceptible to network issues and therefore most projects feature a requirement to enable offline usage of the application.

1.2 Outline

After an overview of the prerequisites in chapter 2, we start with a comprehensive introduction of the problem in chapter 3. Chapters 4 compiles our research process and

[1] There is now an estimated 12+ million mobile application developers worldwide [1].

provides a run-up to our original proposal which is further described in chapter 5. Chapter 6 ties the related work with our research, and finally, chapter 7 summarizes our results and hints at possible future work.

1.3 Definitions

We provide below some of the definitions we use in this thesis. Those are commonly found in the related literature.

Stores are modeled as sets of objects which can be read or written to. Read and write operations are denoted as follows:

- $w_x(a)$: write value a to object x
- $r_x(a)$: read value a from object x

$\perp$ denotes the null value, such that $r_x(\perp)$ means that x is absent from the database. $w_x(\perp)$ removes x from the database.

Transactions are represented as ordered sequences of operations. For instance $T_1 = r_x(1), r_y(a), w_x(a)$ means transaction T_1 reads the value 1 from x, then reads some symbolic value a from y and writes it back to x.

We model **Databases** like stores but where each element contains a vector of values, like rows in a relational database table.

Chapter 2

Background

This chapter provides the background information required to understand the rest of the thesis. Sections 2.2, 2.3 and 2.4 recall the definition of ACID transactions, the CAP theorem and Multiversion Concurrency Control (MVCC), respectively. The next sections define more subjective or recent notions in the context of the thesis.

2.1 Consistency

The word **consistency** may carry a whole range of meanings, two of which play a significant role in this dissertation. Therefore it is essential that we distinguish them as to avoid confusion.

The first meaning, which we alias as **schema consistency**, refers to the following definition.

Definition 2.1.1 (Schema consistency)**.** *A database is **schema consistent** if and only if no rule defined on said database is broken by said database's state.*

In section 2.2 we define the rules involved to give this definition a practical turn.

The second meaning, which we refer to simply as **consistency** or **consistency class**, classifies distributed systems according to the guarantees they offer on reads with respect to past writes. Consistency classes define a set of anomalies which cannot occur, in that regard, it is the same thing as ACID's isolation levels. The more excluded anomalies, the stronger the isolation level. Anomalies and isolation levels have been extensively studied in Atul Adya's Ph.D. thesis [2] which provides a sound framework for subsequent work. We base the following definitions on those provided by W. Vogels [3].

Definition 2.1.2 (Strong consistency)**.** *A database is **strongly consistent** if and only if, after an update to an object, any subsequent read of that object returns the updated value.*

Definition 2.1.3 (Eventual consistency)**.**
*A database is **eventually consistent** if all reads of an object return the same value after some finite amount of time in the absence of updates. This time is called **inconsistency window**.*

2.1.1 Causal consistency

A couple of interesting consistency classes are **causal consistency** and **causal+ consistency**. These are based on the notion of potential causality, denoted $\rightsquigarrow$, and defined by Wyatt et al. [4] as follows.

Definition 2.1.4 (potential causality [4])**.**

1. ***Execution Thread***. *If a and b are two operations in a single thread of execution, then $a \rightsquigarrow b$ if operation a happens before operation b.*
2. ***Gets From***. *If a is a write operation and b is a get operation that returns the value written by a, then $a \rightsquigarrow b$.*
3. ***Transitivity***. *For operations a, b, and c, if $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$.*

Under **causal consistency**, values read are consistent with the order defined by causality. **Causal+ consistency** additionally requires convergent conflict handling, which means all conflicting updates be handled the same way at each replica.

2.2 Transaction properties

Transactions are a fundamental building block of database systems. When operations are grouped in transactions, we can construct applications on logical business operations rather than the low-level operations they contain. In order to be of any use, transactions may provide guarantees which are usually defined as Atomicity, Consistency, Isolation and Durability (ACID) properties [5] [6]. These properties define the behavior of transactions under expected, even if exceptional, conditions such as concurrent execution of any combination of transactions, system crashes and power outages.

The Atomicity, Consistency, Isolation and Durability (ACID) framework does not cover bugs, hardware glitches, and other failures which are assumed to be handled by the lower abstraction layers.

Atomicity guarantees that either all operations in the transaction succeed or none of them do. Once the transaction ends, subsequent observations either see the result of all operations or see no change at all.

Consistency is the **schema consistency** we define in section 2.1. The rules involved in this definition are, among others, **foreign keys**, **triggers**, **uniqueness** and **not-null**[1] which we describe in section 2.5. We do not include the other meaning of Consistency in ACID even though some sources do. Instead, we reserve it for the CAP theorem in section 2.3 where it helps explain our work while minimizing confusion.

Isolation refers to which degree, called **isolation level**, concurrently executing transactions interfere with each other. At each end, we have **serializability**, which guarantees

[1] Although most of these concepts are specific to relational databases, we do not need additional definitions for NoSQL databases as they do not introduce new concepts of schema consistency. They either drop schema consistency altogether or feature some equivalent to some constraints of relational databases.

an equivalence with some sequential execution of all transactions involved, and **read uncomitted** where transactions can read each other's uncommitted data. Typically, lower isolation levels may introduce business level inconsistencies[1] but also provide higher performance. **Serializability** with **strong consistency**, that is, the execution is equivalent to a sequential execution of all transactions and the sequence order is the real-time order of all commits, is called **strict serializability**, an even stronger isolation level [7].

Isolation levels are notorious for being misunderstood and misused. The definitions provided in ANSI SQL have often been criticized and are unsuited to qualify distributed databases [6] [8]. We treat this subject in more details in the next section.

Durability ensures that once a transaction ends, its changes, if any, will not be lost under the expected conditions.

2.3 CAP theorem

Gilbert and Lynch [9] find that "it is impossible to reliably provide atomic, consistent data when there are partitions in the network", in the context of web services. In this statement, consistency is defined as follows: "committed transactions are visible to all future transactions", which is in fact the same as our definition of **strong consistency** (def. 2.1.2). This claim is also called the CAP theorem and applies to distributed systems in general.

The theorem implies that if a system needs to tolerate network partitions, which distributed system ought to do, then such a system cannot be strongly consistent and highly available at the same time. In other words, some requests to the system cannot succeed in bounded time and return a consistent result.

Theorem 2.3.1 (CAP theorem [9])**.** *A distributed system cannot answer all requests with* ***strong consistency*** *in bounded time, shall network partitions occur.*

Note that this theorem does not deal with **schema consistency** and hence doesn't exclude systems which return **schema consistent** results in bounded time. This is an essential remark as we shall see in the next chapters.

Distributed systems are said to be AP or CP depending on whether the design favors availability or consistency respectively. Bailis et al. [10] went further and showed which consistency classes are achievable in a highly available system. They define isolation levels capable of supporting features such as foreign key constraints and global secondary indices in a highly available system [11]. This is a good indication that in fact **schema consistency** is possible where **strong consistency** is.

As can be seen in figure 2.1, the CAP theorem focuses on a small portion of the picture but, due to its popularity and apparent simplicity, it is often misinterpreted as having a broad meaning covering the whole landscape.

[1] Not to be confused with any of the definitions of **consistency** from section 2.1. This one lives on an entirely different abstraction level.

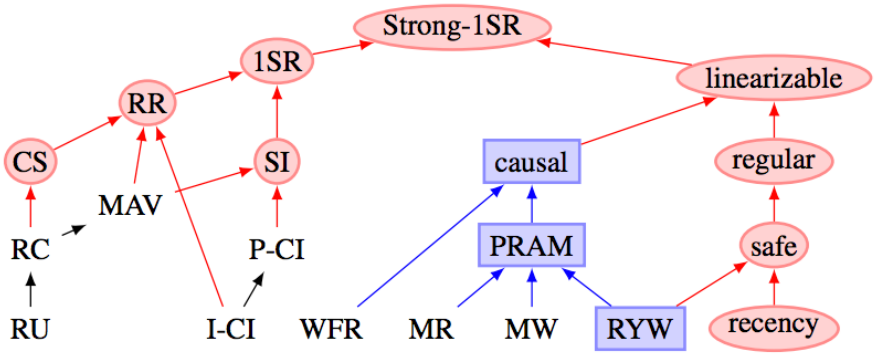

FIGURE 2.1: Classification of isolation levels by Peter Bailis et al. [10]. Some can be achieved in a highly available distributed system (white and blue) and others can provably not. The CAP theorem only deals with *linearizable* (top right).

2.4 Multiversion Concurrency Control

To achieve high levels of isolation, database systems historically used lock-based protocols. For instance, the two-phase locking protocol can achieve serializability [12]. However, this strategy tends to be inefficient in distributed environments [13].

Multiversion Concurrency Control (MVCC) represents a family of techniques which aims at providing the same isolation guarantees as lock-based algorithms with better horizontal scaling capabilities.
The core concept of MVCC is that objects are never modified in-place. Instead each time an object is modified, a new version of that object is instantiated. This means that multiple processes can see different versions of the same object at the same time, hence the name *multiversion*.

The terms *optimistic locking* and *pessimistic locking* are often used to describe the contrast between MVCC and traditional methods although MVCC does not require locks.

MVCC is gaining popularity due to an increase in awareness about the benefits of horizontal scaling in the industry. Nowadays all major Relational Database Management System (RDBMS) vendors offer MVCC-backed isolation levels, often times by default.

MVCC is easily confused with snapshot isolation (SI) (SI in figure 2.1). In this isolation level, each transaction is given an isolated snapshot of the database to work with, which is easily achieved with MVCC. However, an important and counterintuitive feature of snapshot isolation is that it prohibits *lost updates*, that is: if a transaction T1 reads a data item, then T2 updates it and then T1 updates it based on the value previously read, then both transactions cannot commit as T2's update would be lost [6]. Because of this, *snapshot isolation* cannot be implemented in highly available systems.

2.5 Common database rules

Some database features allow the modeling of business rules by restricting the set of possible (i.e. **schema consistent**) database states. Originally from the relational world, they sometimes find equivalents in the **NoSQL** realm.

2.5.1 Foreign key constraints

Foreign key constraints are the cornerstone of relational databases: they express relationships between tables. Some **NoSQL** databases (*e.g.* graph databases) have similar mechanisms which we will not study here as they vary widely between database implementations; however, they are expressing the same fundamental concept as **foreign keys**.

In this work we define a **foreign key** constraint as an invariant involving an attribute from a table T_1 and the primary key of a possibly different table T_2.

Definition 2.5.1. *A foreign key on attribute* a *of table T_1 referencing table T_2 defines the following invariant.*
For each row in table T_1 with attribute a*, there exist a row in table T_2 whose primary key is the value of a.*

This definition allows us to claim the following theorem, which we prove in appendix A.

Theorem 2.5.1. *A foreign key constraint on attribute* t *of table T_1 referencing table T_2 is broken if and only if any of these conditions is met:*

1. *An entity in table T_1 is added or modified and there exist no primary key in T_2 whose value is that of the new value for attribute* t
2. *An entity in table T_2 is removed and there exists an entity in table T_1 whose value for attribute* t *is that of the primary key to be removed*

2.5.2 Triggers

A trigger is a piece of code which performs changes in a database in reaction to other changes. It is mainly used to maintain some invariants which cannot be expressed with regular constraints. Sometimes triggers are used to notify the external world of changes in the database, but this is not relevant to **schema consistency**.

Capabilities of triggers vary widely between database implementations, both regarding what they can do and what they can react to. Some **NoSQL** databases also offer to execute code in reaction to changes although the code rarely executes atomically with respect to the changes that triggered it.

2.5.3 Other constraints

While the reader may know of other exotic constraints, we only mention the two most common ones: **unique** and **not-null**.

The **unique** constraint ensures that no two entities of the same type can have the same value for the given field or combination of fields. This definition has a straightforward mapping to RDBMSs: substitute “row” for “entity”, “table” for “type” and “column” for “field”. NoSQL stores often have unique constraints as well despite the fact that they

may not have a proper schema.

A **not-null** constraint simply ensures that a field may not contain a null value. It becomes particularly interesting when combined with a foreign key constraint, thus expressing a relationship which must exist at all times.

The attentive reader will notice that we have only studied **non-nullable foreign keys** above. However, our results can be generalized to the **nullable** case by merely adding an implicit row in table T_2 with *null* as its primary key.

2.6 NoSQL

A large number of database systems have sprouted under the umbrella term **NoSQL** since around 2009. This term has been caught in a vicious cycle whereby new products associated themselves with the **NoSQL** movement in an attempt to benefit from its hype and consequently added to that movement's hype. As a result, much confusion exists around **NoSQL**: false beliefs are being relayed and untrue claims made.

An example of such belief is that **NoSQL** systems pioneered MVCC. While most if not all new NoSQL databases rely on MVCC by default, relational databases have started implementing it since before the NoSQL era. Notable examples are Oracle RDBMS, PostgreSQL and even MySQL with InnoDB shipped by default as of version 5.5.5 (July 2010)[1].

2.7 Git (version control system)

Git is a popular version control system used by developers to collaborate on software projects. It enables users to work offline and deal with conflicts upon synchronization. In that regard, git is an effective distributed persistence system with offline capabilities, not dissimilar to the solution we are after. We describe some of those solutions in the next subsections. It is worth noting that these concepts are not exclusive innovations of git, but we expect our readers to be familiar with the tool and studying the features in context makes the task easier.

2.7.1 Branching model

In git, the codebase evolves in successive commits. Each commit describes a complete state of the codebase. Commits have one or two parents and any number of children. The full graph of commits describes the history of the codebase. Developers usually work on branches. A branch is a pointer to a commit which moves along as the developer appends new commits or merges with other branches.

The compelling idea from git's branching model is that it makes it much easier for multiple developers to work on the same set of files. Each developer maintains his local branch and when they want to synchronize their changes, git will ask one of them to merge and resolve the conflicts. This should sound familiar when we introduce the solution to the UX workflow in section 3.2.

[1] https://dev.mysql.com/doc/refman/5.6/en/innodb-introduction.html

2.7.2 Three-way merge

Three-way merge is the default merging algorithm in git. This simple yet powerful algorithm relies on two assumptions: the diff between two commits can be efficiently computed, and the commit graph is simple enough that the lowest common ancestor (LCA) between two commits can be found rapidly. The LCA between commits A and B is the closest commit from A or B which is an ancestor of both A and B. The diff between A and B is the set of changes which yields A when applied to B.

To perform a three-way merge between commits A and B, we first search for the LCA L between A and B. We then compute the diff between A and L and apply it to B. Changes in the diff encode what line needs to be replaced and what line replaces it. Git detects a conflict if the source line of a change is different from what is expected or if it can not be found. Figure 2.2 illustrates three-way merge between two non-conflicting commits.

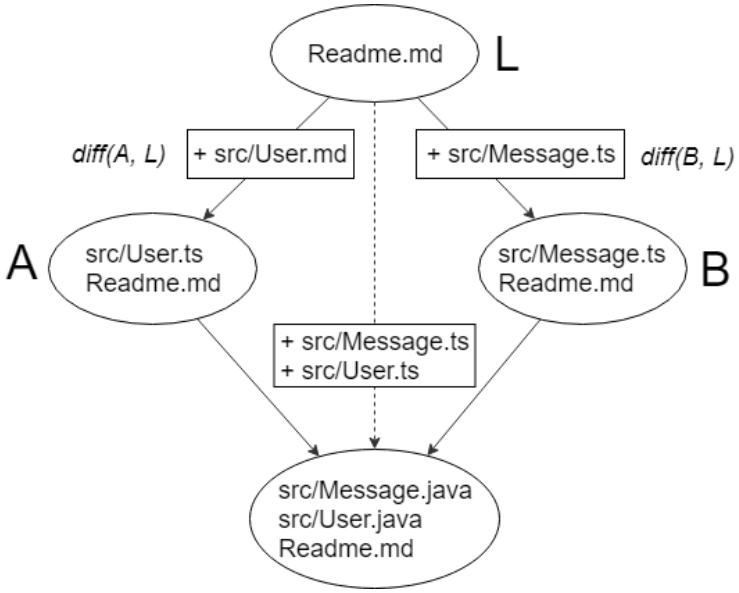

FIGURE 2.2: Three-way merge illustration. To merge commits A and B with lowest common ancestor L, git uses the *diff*s between A and L, and B and L, and applies both on top of L.

2.7.3 Merkle tree

Merkle trees are trees where each leaf node is labeled with the cryptographic hash of its content, and each non-leaf node is labeled with the cryptographic hash of its children's hashes. Comparing and differentiating Merkle trees is efficient because identical subtrees do not need to be visited: if their hashes are equal we know they are structurally equal as well.

Git uses a variation of Merkle trees to efficiently compute diffs between commits. The whole directory structure of each commit is encoded in a Merkle tree. When git diffs two

commits, it efficiently compares their Merkle trees to find which files are different.

Aside from git, many distributed databases such as Riak[1], Cassandra[2] and DynamoDB [14] use Merkle trees as a way to ensure data consistency across replicas.

[1] `https://github.com/basho/riak_core/blob/1.4/src/merkerl.erl`

[2] `http://cassandra.apache.org/doc/latest/operating/repair.html`

Chapter 3

Objectives

Many business-to-business (B2B) projects require the deployment of an application on portable devices which are carried around by employees to fulfill tasks. The perimeter on which said employees evolve ranges from a single warehouse to the whole railway network of a country. In such environments, it is necessary to assume that the network is unreliable. On the other hand, it is vital that the application stay available. Our work focuses on the solution to solve this apparent dilemma.

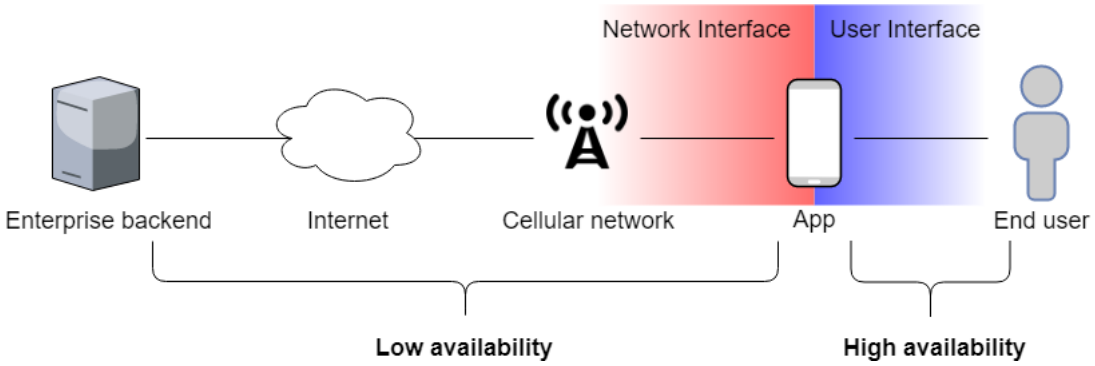

FIGURE 3.1: The mobile app availability dilemma

3.1 A user experience issue

3.1.1 Time barriers

Jakob Nielsen mentioned the concept of response time bounds in *Usability Engineering* in 1994 [15]. Back then the concepts were already thirty years old and the fact they are still relevant today despite the massive technological achievement of the past half-century has something beautiful to it. Three thresholds are defined, **0.1**, **1** and **10** seconds, and roughly mean the following.

A waiting time of less than **0.1 second** is expected for immediate feedback. This includes toggling checkboxes, starting the loading of a new page, haptic and sonic feedback in response to button presses and any feedback which helps materialize the user interface. Longer delay times cause the application to feel glitchy; widgets feel like mere images on a screen rather than real-world entities.

Delays below **1 second** are acceptable for routine tasks such as opening a complex

menu or going to a different screen. It applies to any response to user actions which do not involve obvious processing complexity or are to be frequently performed by the user. Longer delays break the feeling of immediacy and can cause the user to lose his train of thought[1].

Any modal[2] action should take no longer than **10 seconds** and should show some feedback to signal that something is happening. After 10 seconds, users lose interest and start multitasking. Even worse, in the case of enterprise mobile apps, it may prevent workers from doing their job!

Of course these constraints build on top of each other: for instance, when the user hits the search button to execute a complex query, the button should light up in under 0.1 seconds, the search result page should load in less than 1 second, and the results should appear in less than 10 seconds. Should any of these conditions not be respected, the user experience will be negative.

Although delays of more than 10 seconds are forbidden, some tasks require more than 10 seconds to complete. Clever user experience (UX) design goes around this limitation by deferring the work — enter the asynchronous UX workflow.

3.1.2 Asynchronous UX workflow

When a smartphone performs is updating installed applications — which can take several minutes — the user can continue using other features of the device like texting or web browsing[3]. This is an example of an asynchronous UX workflow. In a synchronous flow, on the other hand, the user would have to wait on a modal dialog until all updates have completed; it would not be acceptable user experience.

In essence, a workflow is asynchronous when processing is pushed to the background in order to fulfill UX time barriers. Optionally, the user can be notified when processing is complete; for instance on a mobile device the application may send a system notification. This is how to make responsive and available apps despite network unavailability.

3.2 Solutions for the asynchronous UX workflow

The asynchronous UX workflow is an idea; its implementation requires building blocks. What are those building blocks?

From a high level perspective, each function in an application controller performs *write*, *reads* or both with respect to business data. It follows that an available application must be able to perform reads at all times, which means the data has to be on the device

[1] The relationship between the 1 second bound, "the zone" and the specious present is a fascinating research topic of psychology. The curious reader can learn more about it in *Flow: The psychology of Optimal Experience* [16]

[2] Which blocks interaction with the app until completion

[3] The fact that mid-range devices tend to become unavailable during software updates is an implementation issue rather than a design flaw and shall not disprove our point.

itself. We conclude that there must exist a data store on the device with all relevant business data available at all times.

An attractive architecture results from this observation: since we cannot get rid of the embedded database, instead have the application interact exclusively with that database. The database periodically synchronizes with the backend on a different thread such that it does not interfere with the user interface. As simple as this solution looks from the outside, its implementation presents many challenges.

3.2.1 A challenge

Handling synchronization errors is seen as one of the most painful tasks in the implementation of offline-capable enterprise mobile applications. Except in a few rare cases where clients are working on disjoint subsets of the data, writing such an application is akin to writing a distributed application; the shared state may be modified by different users concurrently; therefore, conflicts are unavoidable.

Although high profile software editors have successfully employed generic methods to solve their synchronization issues in an engineering-friendly fashion [17], most work done in the IT consultancy business relies on ad-hoc, costly and error-prone solutions. This fact could find an explanation in the way these firms work: they tackle various projects, each having its own budget. In such context, it is hard to spot common patterns with a potential for factoring out reusable parts. Given that the development of a generic synchronization channel is laborious, time-consuming and has a high potential for failure, such investment can be perceived as a loss of resources or a gadget which will hardly be amortized over many projects even if successful.

3.3 Requirements of enterprise mobile applications

We formulate a set of recurring requirements for projects involving data synchronization based on insights from ELCA. Contrary to business-to-consumer (B2C) products, B2B applications are usually given complete ownership of the device: there is no need to share storage and compute resources with other applications since the device's only purpose is

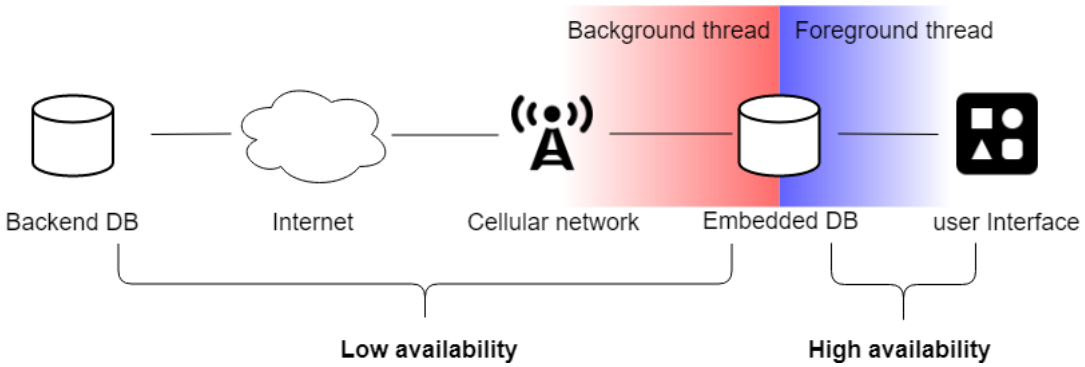

FIGURE 3.2: Architecture of a solution to the mobile application dilemma

to fulfill the business use case. Despite this fact, the data at rest must usually be secured to protect business property. Security is a vast topic and, in the context of this thesis, we merely discuss access control and leave advanced security issues for future work.

Another significant difference is in the number of end users which is usually orders of magnitudes lower: while a general purpose application may target over a million users, typical companies have anywhere between a few dozen field agents to a few thousand at most. Similarly, the business data is generally of moderate size: between 1 and 10 gigabytes in total. It is common practice to archive historical data by moving it to another store such that the primary database only contains active data. This is mostly done to keep the master database responsive, but in the case of enterprise mobile apps, it also helps reducing bandwidth and device memory usage.

The systems we study are generally single components of complex enterprise infrastructures, and therefore such systems must obey the constraint of the infrastructure, not the other way around.

Given these observations, we establish the following list of requirements.

- The architecture is solely a client-server architecture. Previous work on peer-to-peer (P2P) architectures [18] identified potential gains when the agents form interconnected clusters. In our general use case, the weakest link in the communication chain is the one closest to the agents (the cellular network); bandwidth, availability, and latencies beyond that point up to the backend are less worrying. We focus this work on making the single backend more scalable and available and leave P2P aside.
- An overwhelming majority of business data is stored in SQL databases. Therefore we assume that our system will have to interact with master data located in a SQL store, or that our system provides a SQL interface with qualities comparable to those of SQL databases currently in use in the industry. Non-compliance with this criteria would drastically restrict the addressable market. In short, the consolidated data must be persisted to a SQL database on the backend.
- While we do not deal with authentication and authorization in this work, we must nonetheless show that an access control mechanism could be built on top of our system. In particular, we want to be able to restrict which data users can write and read.
- Previous projects at ELCA ran into battery autonomy issues. Also, we want to avoid expensive computations on the mobile device to save as much power as we can.
- Similarly, data transfer costs power, time and money. Therefore we want to limit the amount of data leaving and entering each device.

Chapter 4

Investigation

In this chapter, we outline our exploration of the design space. Multiple designs were evaluated through proofs of concept; some were aborted when they revealed significant limitations. Each attempt provided a different angle of attack which allowed us to characterize the many technical challenges that arise when implementing synchronizable databases.

We start in section 4.1 with a solution based on CouchDB, a popular NoSQL document store, and show that it is not suited to relational data. In section 4.2 we move on to an implementation based on off-the-shelf SQL databases and HTTP, and we discuss many pitfalls and their possible workarounds. The design from section 4.3 is an original idea which stemmed from the shortcomings of the previous attempt and is also based on existing SQL databases.

We close this chapter with section 4.4 and an introduction to our most ambitious proposal, a relational database engine built from the ground up to support our usage scenario.

4.1 CouchDB

CouchDB[1], a schemaless multi-master document store, and its client-side JavaScript complement PouchDB[2], "The database that syncs" [*sic*], form a trending couple in the scene of progressive web applications. They implement a flexible synchronization protocol, hereafter the *CouchDB (synchronization) protocol*, which allows application developers to synchronize data across many clients effortlessly. There are multiple document databases which implement the CouchDB protocol, among which Couchbase[3] is worth a mention. Couchbase features a program called *Sync Gateway* which implements the server side of the CouchDB synchronization protocol. *Sync Gateway* allows application developers to write a small javascript function dubbed *Sync Function* to filter and transform data as it is synchronized. This function can be used to enforce access control and limited data validation.

In our experimental setup, we use *Sync Gateway* with PouchDB. Note that Couchbase also offers an embedded database compatible with the CouchDB protocol which one could use *in lieu* of PouchDB. Also, in the remaining of this discussion we use CouchDB

[1] `http://couchdb.apache.org/`
[2] `https://pouchdb.com/`
[3] https://www.couchbase.com/

interchangeably with Couchbase Sync Gateway where it does not matter which implementation is used. Figure 4.1 shows each component in a typical setup.

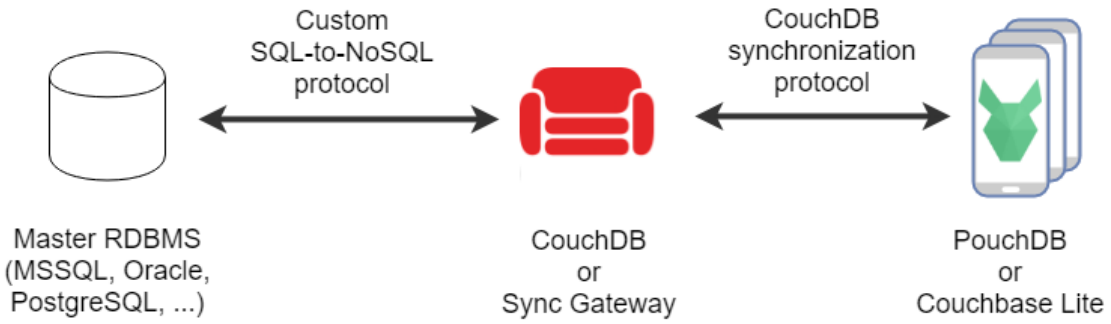

FIGURE 4.1: A typical CouchDB setup backed by a SQL database.

4.1.1 Features

Out of the box, the Couchbase Sync/PouchDB setup fulfills most of our requirements:

- a client/server architecture with a horizontally scalable server
- a mechanism for access control (through the *Sync function*)
- a client database optimized for embedded environments
- a synchronization protocol designed to minimize bandwidth usage

We performed benchmarks to verify the compatibility regarding performance and got encouraging results. More details on these benchmarks can be found in appendix B. The only drawback so far is that all databases we found which support the CouchDB synchronization protocol are *schemaless*, i.e., they feature no or little **schema consistency** rules.

We intend to deploy Couchbase Sync in front of the master SQL database to absorb the high fan-in traffic from the mobile fleet, handle synchronization, conflict resolution, and access control. A synchronization agent sits in the backend between the master SQL database and Couchbase Sync to exchange data between them. A thin interface in front of the client database has to be implemented to enforce **schema consistency** manually upon updates and the *Sync function* may also have to be designed in a way to preserve such consistency during merges.

4.1.2 Integration

Our job is to create a piece of middleware to interface the CouchDB server with the master relational database. There are two ways we can map the SQL schema to documents: aggregate related entities into self-consistent documents (a) or map each SQL row to a single document (b).

The first strategy does not resonate well with our requirements for multiple reasons. First, it requires application designers to find ideal cut points in the relational schema to segregate data into documents. Coming up with such mapping is challenging and

depends entirely on the business; there is little potential for factoring out generic code. Secondly, to create useful aggregates, one needs to know how the data will be used and presented on the front-end. This has the undesirable side-effect of tightly coupling back-end code to the front-end.

As an example, imagine a schema with users and departments. A user belongs to exactly one department. At first, the application only needs to show individual user profiles. It seems reasonable to create *user* documents which aggregate everything we know about individual users, including their respective department. Now the client asks that the application also show the list of users per department. The way we aggregate the data does not allow us to perform such query efficiently. To fulfill the client's request we now have to change the backend code to aggregate documents in a more suitable way, flush data from all agents and re-synchronize the whole dataset across the complete fleet of devices.

The second approach seems more flexible. Each row is stored as an individual document whose key is that row's primary key. With this mapping, we can emulate foreign key relationships by storing the referenced document's id in an attribute of the referrer. Both scenarios from the example above can be achieved with this data with reasonable (sublinear) performance. Additionally, we do not waste space with duplicated data as was the case before.

There remain a few drawbacks compared to an embedded SQL database: the application developer needs to construct queries manually and cannot rely on high-level features of SQL such as joins and complex WHERE clauses, and more importantly, **schema consistency** is not enforced.

We chose strategy (b) to avoid the client-server code coupling issue.

A simple shim is added on top of the store's client-side API to enforce **schema consistency**. As we show in theorem 2.5.1, a foreign key violation can be checked quite easily whenever a document is created or removed. **Not-null** and **unique** constraints can also be efficiently checked with a lookup, given proper indexing (which PouchDB offers). We create a special client, dubbed *synchronization agent* which listens to changes in CouchDB, converts the changes to SQL INSERT, UPDATE and DELETE statements and applies them on the main DB. We modify the database schema such that all tables contain a *last_modified* field which changes each time the data is touched[1]. The first time the system is brought up, the synchronization agent iterates over the whole database and converts all rows to NoSQL documents which it saves in CouchDB. We call this phase the *initialization phase*.

4.1.3 Limitations

Two-way synchronization

Our system loads the contents of the master database in CouchDB once and then keeps pushing updates from the master database to CouchDB. In most scenarios, it is desirable that the changes in the SQL database (due to external systems sharing this database) be replicated to CouchDB. Capturing changes live in the database can easily be achieved using implementation-dependent features such as MySQL's binary log[2] or triggers in

[1] This is common practice in the industry and in a typical integration scenario this field is already present.

[2] `https://dev.mysql.com/doc/refman/5.7/en/binary-log.html`

Oracle Database[1]. Even without such features, one could poll the database for rows with *last_modified* greater than the greatest *last_modified* value seen so far[2].

The real challenge is to find a way to distinguish changes caused by the synchronization agent from those caused by unrelated systems using the same database. If the agent detects its own changes and replays them, a positive feedback loop will cause more and more changes to loop through the system until resource exhaustion. Although this problem is not unsolvable, to our knowledge, all solutions involve grotesque hacks. We only outline two of those solutions here for illustration purposes and spare the details.

One such solution consists in storing all outbound changes near the synchronization agent such that it can then filter out inbound changes it caused. Another solution is that the synchronization agent tags SQL rows using a special column. This tag indicates that the synchronization agent performed the last update to that row. Both solutions have many shortcomings which we do not detail in this discussion to keep the level of detail reasonable.

On the other hand, this problem finds an elegant solution if we relax some design constraints: if the synchronization service has exclusive ownership over its database, then other services wishing to consume from or push data to this service must use that service's API. Preventing direct access to the database allows the service to catch changes made by external actors. In our specific case, the API can be implemented like a regular client in the device fleet, thereby resolving the issue of two-way synchronization.

It is seldom feasible to reroute all services in an existing infrastructure when introducing a new component. However, the advent of *microservice architectures* in recent years is playing in favor of non-shared databases[19].

Foreign key integrity

Data integrity is enforced when a client writes data thanks to the shim mentioned in section 4.1.2, and when the **synchronization agent** applies changes to the RDBMS. We want to ensure that **schema consistency** is preserved throughout synchronization. Otherwise, the RDBMS would reject inconsistent updates.

From a business standpoint, it is preferable to resolve integrity issues when data is first created (we still know the user's intent) or during a merge (we have access to old revisions of the conflicting documents). There is little information left to take conflict resolution decisions when data reaches the synchronization agent. An automatic resolution could lead to violations of business invariants and accumulation of errors.

Take for instance a social network where a user creates a profile and then immediately adds a post. If the synchronization agent sees the post before the user profile, the foreign key *post -> user* will break and the RDBMS will reject the attempt to save the post. If other users comment on that post in the meantime, the foreign key *comment -> post* and the missing post will prevent the comments from being saved. We can see that inconsistencies could quickly propagate through the relational graph and cause major quirks, even though in reality no user ever violated any business rule locally.

The CouchDB changes feed is the sequence of the latest known document states in order of last modification. This means that a write pattern $w_x(1)w_y(2)w_x(3)$ can in fact

[1] `https://docs.oracle.com/cd/B28359_01/appdev.111/b28370/triggers.htm`

[2] We do not discuss the risk of race conditions on *last_modified* to keep the argument focused.

appear in the changes feed as $w_y(2)w_x(3)$ (previous writes to x were shadowed by the last write to x).

We experimented with ways to circumvent this issue, but we quickly realized that it was akin to designing our own replication protocol on top of CouchDB's, relying merely on its message delivery properties. During this process, we conjectured that the synchronization channel must preserve some sort of causality relationship for foreign key constraints to survive. We later found confirmation of this hypothesis in the literature: causality [4] is required to enforce foreign key constraints and some other essential features of relational databases [10], [11].

4.2 SQL-to-SQL

We move on to a different approach building on simpler primitives. In this attempt, we establish a protocol to synchronize SQL databases embedded in the clients with the master SQL database over HTTP. We first implement a naive solution to see what issues arise, and then we try to address each of them.

We need a way to retrieve changed (added, modified or removed) records on the client so we can send them to the server. Conversely, we want to send the client the difference between what it currently knows about the master database and the current state of the master database.

An indexed column *updated_at* is added to each table. Its value is set to the current timestamp whenever a row is updated. The client remembers the timestamp of the last synchronization and therefore can select all rows in its local database which changed since the last synchronization. When synchronization is initiated, the client reads all changed records in ascending order of the *updated_at* column and sends them to the server. The server replays those changes on the master database; then it reads all records with an *updated_at* column greater than the last synchronization timestamp and sends them back to the client.

4.2.1 Issues

We can already identify potential issues with this solution.

- Foreign key constraints may break on the server.
- The client and server datasets may drift apart.
- The clocks on the server and the client need to be synchronized. Daylight saving time can cause troubles with ordering if the systems do not use universal time.
- Race conditions may appear if a client finishes fetching changes during the same millisecond another client persists a change on the server.
- Conflicts are not detected, and a last-write-wins policy automatically applies which is not suitable for many use-cases.
- Row deletions are not synchronized.

4.2.2 Remediation

We refine the above algorithm to overcome these problems.

Race condition and clock skew

Instead of a timestamp, we use a sequence id and name that column *seq*. The sequence is a per-table unique auto-incrementing value.

The next sequence id for each table is included in the initialization payload. When a client synchronizes, it fetches all records with *seq* greater than or equal to the last known sequence value for each table (i.e., all records that changed since the last synchronization). The server saves all those changes and assigns them new sequence numbers according to its internal counter. The server then sends all updated rows since the last sequence known by that client and includes the new sequence counter for each table.

Using a global sequence number gets rid of the potential race conditions and other issues related to timestamps. The counter effectively acts as a coordination point which guarantees correctness but prevents horizontal scalability [20].

Synchronize deletions

If a client deletes a row, then all information about that row is lost, and the server will never know that the row was removed. This problem can be solved using *soft deletion*.

Soft deleting a row refers to setting a particular column to some value such that application logic excludes the row in all queries. In our implementation, we create a column *deleted* which indicates that the row is deleted when its value is not null. With this technique, deletions are the same as updates which we already know how to handle.

The downside of this approach is that a job must periodically run to remove all rows which have been deleted. In practice, however, this is not a problem since most applications already use some a *soft delete* mechanism.

Client-server data skew

If a client makes changes, sends them to the server and if the server skips some changes (due to conflicts for instance), then the client and server states may start drifting apart. We explain two different approaches to solving this problem.

One way to solve the problem is to make sure the server returns not only the rows which have changed since the last time the client synchronized but also a difference between what the client sent and what was applied on the server. This way the client first receives a correction of its changes and then the changes made by other clients.

The second way is that the client stores an undo log for each operation it performs. Before the client applies the changes sent by the server, it rollbacks all local operations, thus ensuring that the state following a replication is always the server state.

The second approach has the benefit of a simpler model to implement and analyze, with less potential for bugs, at the cost of increased CPU and storage usage. That being

said, in the scenarios we study changes made by the client represent a negligible proportion of the overall changes: in a fleet of 1000 devices with similar usage patterns, one client's changes represent roughly one 1000th of the overall changes.

Foreign key constraints

Foreign key constraints may break during synchronization. This is the same problem we faced with CouchDB in section 4.1. Because we use state-based replication — only the state is sent, not the sequence of events which lead to it — we lose the guarantee that foreign key constraints are respected at all times.

The solution we implement is to keep a log of all operations applied to the client database and to send this log instead of sending the changed database state, thereby preserving causal ordering. Of course, this comes at the expense of additional storage usage on the client and a slight increase in network traffic.

Conflict detection

In order to detect conflicts, an integer column *prev_seq* is added. This column is set to the previous value of *seq* when a record is updated by a client, or to *null* if the record was newly created. The *prev_seq* value is updated at most once between each synchronization such that it is always set to a value which was returned by the server (consecutive updates are effectively squashed and seen by the server as single atomic updates).

The SQL pseudocode for setting the value of *prev_seq* is given below.

```
prev_seq = SELECT seq FROM :table
           WHERE id = :id AND seq < :last_known_seq;
```

Where *:table* and *:id* are the table and primary key of the row to be updated and *:last_known_seq* is the next sequence id for this table at the time of the last synchronization. If the result set is empty, we set *prev_seq* to null.

When the server applies a client change, it uses *prev_seq* to detect conflicts like this:

```
if prev_seq = null:
    INSERT INTO :table VALUES(...);
else:
    UPDATE :table SET ... WHERE id = :id AND seq = :prev_seq;
```

The insert fails if another client created the row and the update fails if the *seq* attribute of that row changed since the last time the client synchronized.

With soft deletes, foreign key violations can go unnoticed. To detect them, we store the sequence number of each row referenced by a record in that record's changelog entry. We then detect conflicts with a similar WHERE trick as shown above.

Conflict resolution

We consider three kinds of conflicts.

- **update-after-update**: Two clients update the same row and then synchronize.
- **lost-dependency** (theorem 2.5.1, 1.): A client saves a row which references a deleted row.
- **extra-dependent** (theorem 2.5.1, 2.): A client deletes a row which is pointed to by another non-deleted row.

There are no **update-after-delete** conflicts because using soft deletes allows us to assimilate deletions as updates. When a conflict is detected, we hand over control to a user-provided function. The function is given the current and candidate rows. It can query the database and should apply business specific logic to resolve the conflict.

One drawback we identify with this method is that the conflict resolution function executes out of the context of the original user intent. Say there are two user workflows which may lead to the modification of a specific data item. There may be complex logic within the client to handle edge cases, but the conflict resolution function is only given the one conflicting object and does not know *a priori* from which workflow the modification came.

Another drawback is that if the conflict handling function reads the database to figure out how to resolve the conflict, then the outcome will be highly dependent on the isolation level set on that database. If other transactions happen concurrently, the user code is prone to race conditions.

4.2.3 Performance and correctness

We want the replication process to be atomic. That is, either all client updates are applied, or none are. This prevents replaying the same change multiple times if a synchronization aborts in the middle. We achieve this by wrapping the whole synchronization process in a single SQL transaction on the backend database. The problem is that popular databases tend to perform poorly with concurrent long-running transactions under high isolation levels. This is because transactions need to be aborted and retried in case of a conflict. In the worst case, if a transaction contains many updates and only the last one conflicts, the whole transaction needs to be restarted. Therefore, we expect this solution to suffer significant performance hits when conflicts are frequent.

4.3 Repeatable business transactions

This design implements a different protocol based on the same primitives as the previous one. With this design we seek an answer to the question “What does it take to preserve the business meaning of a change down to the master database?”.

4.3.1 Concept

We start with the enhanced design from section 4.2 and change a few things.

First, we introduce the concept of business transactions (BT). A BT is a piece of business logic which performs modifications on the database. We enforce that all modifications are made through business transactions, that is all database mutations must happen inside of these custom functions. When a client executes a BT, it saves the id of the BT, and any parameter passed to it. The set of BTs forms a model which is shared between the clients and the backend. Listing 4.1 shows two sample BTs.

Instead of sending updates, the client sends the log of all BT executions. The server then executes that log to replicate the client's state. The server then replies with the set of changes effectively applied to the master database.

This solution eliminates conflicts since it eliminates synchronization. Temporary changes on the client provide a preview of the effects of the BTs but the later actually executes on the server. Concurrent synchronization of clients may still cause errors, although these are dealt with immediately within a BT, with knowledge of the business intent.

This concept is appealing from an application developer's standpoint because the same code handles user actions and conflict resolution. BTs are highly decoupled from each other which should ease debugging and lower the overall complexity of the code.

4.3.2 Limitations

Imagine an application with two business transactions. The first one takes a username as input and creates a user with that name, the second one takes a user id and a text string as input and creates a post by that user.

```
create_user():
    let user = new User()
    let id = get_metadata("user_id")
    user.id = id
    user.save()
    return user

create_post(userId, message):
    let post = new Post()
    post.message = message
    post.author = userId
    post.save()
    return post
```

LISTING 4.1: Example business transactions

For this application to work offline, the second BT needs to be able to operate with a user which was not yet synchronized with the backend. We face a problem if the user id is generated in the first BT. If the server generates a different id for the same, then what should be the *id* parameter passed to the second BT? We need to either make sure that the user *id* generated on the server is the same as the one generated on the client, or track

the dependency between the first and second BTs such that, upon synchronization, the second receives the server-generated *id* and not the id it received on the client.

The second option guides down a rabbit hole of symbolic execution and dependency tracking where we would rather not go. The first option seems easier to reason about and implement. We add the following methods to the business transaction's programming interface:

- *set_metadata(key, value)* can be called in the function to save a value which is stored alongside the BT.
- *get_metadata(key)* returns the value set with *set_metadata(key)* when the BT is replayed on the server.

```
create_user():
    let user = new User()
    let id = get_metadata("user_id")
    if id != null:
        user.id = id
    user.save()
    set_metadata("user_id", user.id)
    return user
```

LISTING 4.2: Business transaction "create_user" with id memoization

When the BT is executed on the client, the id is created and saved in the metadata. Then when it is replayed on the server, the id is retrieved from the metadata and forced onto the user. This assumes that we can force the database to use a specific primary key for a row, which prevents the use of AUTO_INCREMENT primary keys, and that the id generated by the client is unique. These assumptions are not absurd, especially given that AUTO_INCREMENT columns are hard, if not impossible, to achieve in distributed environments.

This addition considerably complicates the programming model. An application developer could easily forget to use the metadata as prescribed above and this would result in hard to debug inconsistent behavior.

Other noteworthy drawbacks include the following:

- The server has to execute custom code for each modification even if it is unlikely to cause any conflict.
- This does not solve the issue of synchronizing changes from the RDBMS to the client, such as changes made by external services.
- The API must be built in a way that makes it impossible to write to the database from outside a BT.
- A too high database isolation level could penalize performance, and a too low level would make it hard to code correct BTs.

4.3.3 Performance

This system inherits the performance characteristics of the design it is based on (see section 4.2.3). Besides, tracking which business transactions ran and which did not in the

event of a mid-synchronization crash is tedious. Wrapping all BTs of a synchronization batch in a SQL transaction to benefit from atomicity would kill performance.

4.4 Causally consistent diffable store

Each solution studied so far has advantages and disadvantages of its own. However, none of them provides good parallelism with strong isolation and schema consistency at the same time.

In this section, we attempt to design from lower level primitives a system which addresses the specific objectives from chapter 3. We want a highly available system able to synchronize a relational schema across a large fleet of mobile devices. The system should gracefully handle multi-versioning and delegate conflict resolution to business-specific custom logic. Besides, the load caused by merge operations should distribute on multiple CPUs and if possible multiple nodes in a network. Finally, the database should converge to a consistent state.

This section provides a global overview of the main design principles; An in-depth description is given in chapter 5.

4.4.1 Principle

We model the database state as a graph of immutable commits, similar to the way git works. Each commit describes a complete, **schema consistent** database snapshot. A commit has one or two parent commits (except the root commit which has none) which form an acyclic directed graph reflecting the chronology of the database states.

Data is stored in an immutable, ordered key-value store to benefit from structural sharing and copy-on-write semantics. Immutability also brings snapshot isolation at the expense of garbage collection.

When a client synchronizes its changes with the backend, it takes a commit as input and produces a new one as output. Commits are never modified in place. It follows that concurrently synchronizing clients naturally create divergent branches. The database asynchronously merges branch tips until there is only one left.

Another key point of this design is the way conflicts are detected and handled. First, this system detects not only single-document conflicts (**update-after-update**) but also enforces schema semantics like foreign keys (**lost-dependency** and **extra-dependent** as defined in section 4.2.2). Second, conflict handling is delegated to a user-provided function which runs in an interactive database transaction. This design removes the need for clients to perform rollbacks or integrity checks during synchronization, effectively offloading part of the work from the client to the server. We aim to make the system fault-tolerant, scalable and free of races by using a bottom-up rational approach rather than trying to fit existing systems in a box where they do not belong.

Chapter 5

Implementation

This chapter provides an in-depth development of the idea introduced in section 4.4, with implementation hints and a critical analysis of the key concepts.

We first look at the overall architecture and then we explain how the key components work. Then we summarize the limitations and present tentative ways to improve the design.

5.1 Architecture

The server-side database is built around the following primitive: an ordered, immutable, key-value store. A proper definition of this phrase is provided below. In the following discussion, *efficient performance* denotes a sub-linear expected asymptotic run time with respect to the number of entries in the store.

- **Key-value store**: A persistence class with methods *get(key)* and *set(key, value)* to respectively retrieve and set a value associated to a key.
- **Immutable**: An instance of the store is guaranteed to always have the same internal state. Mutators return a modified copy of the store and do not modify the store in-place.
- **Ordered**: Keys are totally ordered, and in-order iteration from any key is *efficient*.
- **Diffable**: A **diff** between two stores A and B is a set of operations which, when applied on A, yields B. Such object can be *efficiently* obtained for any two stores.

Figure 5.1 presents how components interact with one another in the system. A relational engine implements the relational schema on top of the key-value store. This component is responsible for enforcing **schema consistency**. A registry keeps track of the commit graph and continually merges competing commits in an attempt to reconcile the database state. The synchronization server uses the commit registry and the relational engine to apply changes from clients and to get the diffs required to synchronize clients.

The client-side database is similar to the server but a little bit simpler. Instead of a complex commit graph, client databases only contain a collapsible stack of commits. They feature the same relational engine as the one found on the server. , but this one is simpler as it does not need to be immutable and is backed by persistent storage.

The business model consists of schema definitions, describing schema integrity rules, and access control rules, describing who can write and read what. It is shared between the client and server such that **schema consistency** rules are identical on both sides.

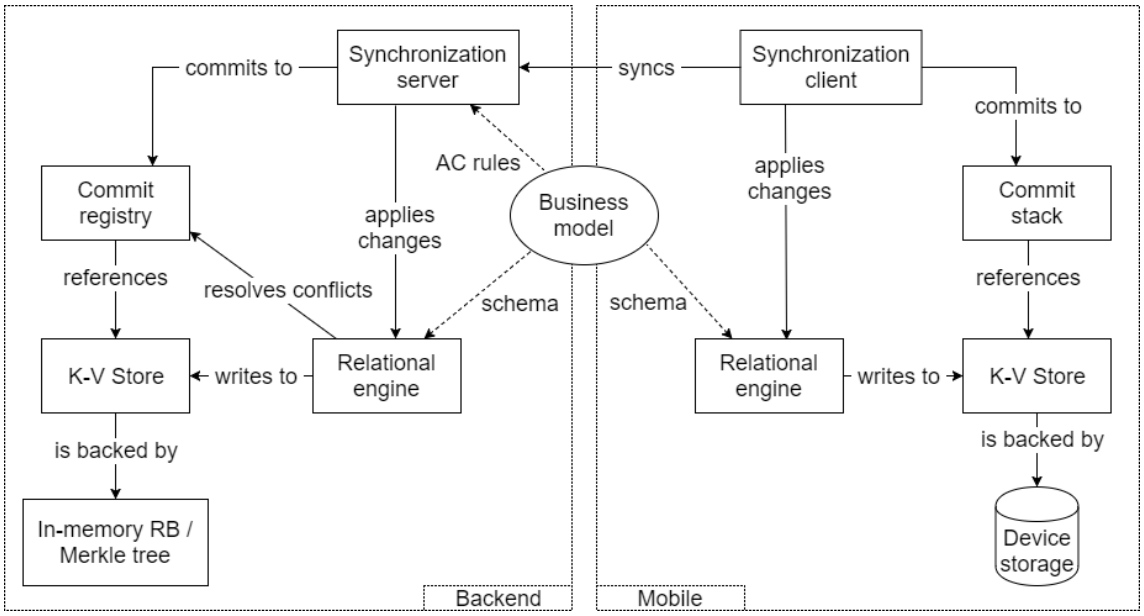

FIGURE 5.1: Architecture of the solution

5.2 Store implementation

In this section we describe the implementation of the key-value store primitive to explain how it achieves the requirements: sublinear access and write time for any element, constant time in-order iteration starting from any element, sublinear differentiation time and an immutable interface with proper structural sharing under the hood.

The functional red-black tree described by Okasaki [21] is an immutable implementation of a red-black tree. It provides logarithmic access and update times for individual items. Since it is a binary search tree, it also enables (amortized) constant time iteration from any element.

As-is, however, it does not provide an efficient diffing algorithm. To this end, we use a Merkle tree [22], another data structure optimized for differentiation.

We create a composite data structure which contains both Okasaki's red-black tree and a Merkle tree. Each node of the red-black tree has a pointer to a leaf of the Merkle tree and *vice versa* such that there is a one-to-one mapping between both data structures[1]. The red-black tree is used for accesses by key while the Merkle tree acts as an index to speed up diff computation.

It is worth noting that those data structures are described as in-memory structures, we discuss ways to persist them to durable storage in section 5.6.1.

[1]We do not combine them in a single data structure because in Merkle trees data is in the leaves unlike in red-black trees. Also, it is easier to analyze the system if it uses off-the-shelf data structures rather than custom ones.

5.3 Mapping the schema

This section describes how we implement relational features on top of a key-value store. Note that we do not implement a full RDBMS but only an engine capable of modeling a relational schema. Features such as SQL query parsing and planning are not addressed. Such engine could be used directly in a project with a thin custom object-relational mapping (ORM) layer on top or could be interfaced with a production SQL database, for instance as a MySQL storage engine[1].

The SQL standard defines many features, and it would be foolish to aim to implement all of them in the first implementation of a new concept. Instead, we focus on a subset of essential relational features which differentiate our solution from non-causal/non-relational stores. The following features are available:

- Basic column types: string, and 32bit integers and floating point numbers
- Simple foreign key constraints without *ON DELETE / ON UPDATE*
- *NOT NULL* and *UNIQUE* constraints
- Indexing of individual columns in natural order

In addition, we assume that table and column names, as well as primary keys, are formatted in such a way that they do not clash with our internal key formats. This can easily be achieved with proper input sanitization or by restricting identifiers to the following character class: *[a-zA-Z0-9_-]*.

—

The schema is defined as code and is shared between the client and server. Other possibilities for storing the schema definition are discussed in section 5.6.2. Each row is serialized using JSON and stored under the key *"table:id"* where *table* is the name of the SQL table and *id* is the row's primary key.

Secondary indices entries are stored as individual documents with key *"table:column:value:pk"*, where *table* is the table name, *column* is the indexed column, *value* is the value of the indexed column and *pk* is the value of the primary key of the row. The value doesn't matter and is set to an empty string.

Foreign key constraints are checked whenever a candidate document is about to be saved. We base our handling of **foreign key** constraints on theorem 2.5.1.

Condition 1. is checked for each of the candidate's attribute upon which a **foreign key** constraint is defined. We can efficiently check the existence of the related entity using a primary key lookup.

To check Condition 2. efficiently, we need to index all columns of a table upon which a **foreign key** constraint is defined. Using knowledge of the schema, whenever an entity is removed we lookup all foreign keys pointing to the table of the entity to remove, using secondary indices for speed. If a result is found, the constraint is broken.
Both checks assume that the data they are working on cannot be modified by concurrent processes which is always true since they work on immutable data.

Unique constraints are enforced at write time by checking that there exists no other record with the same value for that column, using a secondary index for speed.

[1] `https://dev.mysql.com/doc/internals/en/custom-engine.html`

5.3.1 Query performance

We analyse the asymptotic performance of the system for reads and updates.

The most straightforward case is that of the selection of elements by strict equality with a primitive value. There are three cases to consider: whether the filter attribute is the primary key, a secondary index column or is not indexed at all.

In the first case we construct the search key in $O(1)$ time, then we read the object at that key in the store in $O(log(n))$ time where n is the number of documents in the store (rows and indices). We then deserialize the object and return it. Assuming documents are sufficiently small that deserialization cost is negligible compared to lookup cost, we obtain a total asymptotic complexity of $O(log(n))$ for lookups by primary key.

If the search column is in a secondary index, then the cost of lookup is the cost of retrieving all k primary keys that match the given predicate, plus the cost of fetching k records by primary key (which we already know to be $k \cdot O(log(n))$). The lookup by secondary index is actually a range query for keys between *"table:column:value:*α*"* and *"table:column:value:*ω*"* where α is a value less than the smallest character allowed in a primary key and ω is a value greater than the highest character allowed in a primary key. The time complexity of such query is, therefore, the same as the time complexity of range queries by secondary index, which we show below to be $O(log(n) + k)$.

If the key is not indexed, an exhaustive search on all rows of the table is required. This is actually implemented as a range query over all keys between *"table:*α*"* and *"table:*ω*"* with a filter to retain only the rows which match the predicate. Therefore, the time complexity of such query is $O(log(n) + K)$ where K is the total number of rows in the table in which the search is performed.

Range queries work by looking up the smallest value greater than or equal to the lower bound and iterating over the next values until either the value is greater than the upper bound or the end of the store is reached.

If the lookup key is the primary key, then this step yields the results in $O(log(n) + k)$ time where n is the number of records in the store, and k is the number of records matching the query.

If it is a secondary key, the search step yields the ids of the matching rows in $O(log(n) + k)$ time, and an extra lookup is required to get the data of each matching record. The total time for a range query by secondary index is therefore $O(log(n) + k + k \cdot log(n)) = O(k \cdot log(n))$.

If the lookup key is not indexed, then again a full table scan is required with time complexity $O(log(n) + K)$ where K is the total number of rows in the table in which the search is performed.

To update an item one needs to change the item as well as all indices of columns that changed. Updating the data item has the same time complexity as querying it plus $O(log(n))$ to reconstruct the trees (red-black and Merkle). Updating indices is akin to a primary key lookup plus the trees reconstruction step. Therefore if k indices have to be updated, the time complexity is $O(k \cdot log(n))$. Overall the time complexity for updates is $O((k + 1) \cdot log(n))$ for an update affecting k indices.

5.3.2 Transactions

The database is accessed *via* commits, immutable snapshots of the state. Initially, there is a single, usually empty, commit called the root. A branch is defined as the set of commits made of a commit with no children and all of its ancestors. The tip of a branch is the commit with no children within a branch.

A transaction can be seen as an impure function taking a database snapshot as input and returning another snapshot as output. The transaction cannot modify the input snapshot. When a transaction commits, it saves a new snapshot in the database as a commit whose parent is the input commit. When a transaction aborts, it releases references to the work in progress which eventually gets garbage collected. By design, effects of a transaction are not visible from outside that transactions until it commits.

Concurrently executing transactions will naturally tend to create divergent branches (they each create a different child for the same commit). To compensate for this effect, the database continuously performs pairwise merges of branch tips. When there is only one tip left, the database is consistent, and the tip is the reconciled database state. Our database must ensure convergence in bounded time in the absence of updates to provide eventual consistency.

In the following subsection, we analyze the ACID properties of our transactional model. Convergence is treated in section 5.5.

Atomicity

If a transaction aborts, none of its effects would have been visible to other transactions at any instant. Hence an aborted transaction respects atomicity.

If it commits, then its changes are visible only within the output commit or children of that commit. The output commit contains all changes; therefore, a transaction is atomic from the perspective of other transactions starting at the output commit. A merge operation may very well undo some of the effects of a transaction and apparently break atomicity. However, if we conceptualize merge operations as transactions, which they are in practice, then a merge may shadow some of the effects of a transaction without it breaking atomicity (all effects are visible, only some were overwritten by a later transaction).

Consistency

As we have seen in chapter 2, consistency in ACID refers to **schema consistency** and we made the claim that **weak consistency** does not prevent **schema consistency**. Operations within a transaction or a merge operation, which is just a special type of transaction, have to go through the schema enforcement layer. It follows that each database snapshot contains a consistent database state, provided that the schema enforcement layer does its job of enforcing **schema consistency**.[1]

[1]This is a beautiful illustration of the distinction between CAP and ACID: we demonstrated **schema consistency** and yet did not talk about **strong consistency**.

Isolation

The isolation level offered by this design is close to SI. In this non-standard isolation level, transactions operate on exclusive snapshots of the database, much like our commits. One key difference however is that our system allows lost updates (also known as *phenomenon A5B* [6]) which SI forbids. We do however detect such occurrences and allows application developers to act upon it asynchronously with the conflict resolution function.

As far as ANSI SQL is concerned, our design sits between *Repeatable Reads* and *Serializable*. *Repeatable Reads* follows directly from the SI-like behavior described above. The following example proves that our system is not serializable.

Proof.

Given two transactions $T_1 = r_x(a)w_x(a \cdot 3)$ and $T_2 = r_x(a)w_x(a \cdot 2)$ and an object x with initial value 1, if both execute concurrently, the database ends up with two branches; one with $x = 3$ and one with $x = 2$.

We do not know *a priori* what the conflict resolution will do (since it is user-provided) so we assume it is defined as $f(x, y) = x + y$. The reconciliated database state is $\{x = 5\}$ which cannot result from the serial execution of T_1 and T_2. □

Durability

As-is, our system does not provide durability because data is stored in memory. We explore ways to ensure durability with disk persistence in section 5.6.1.

5.4 Synchronization protocol

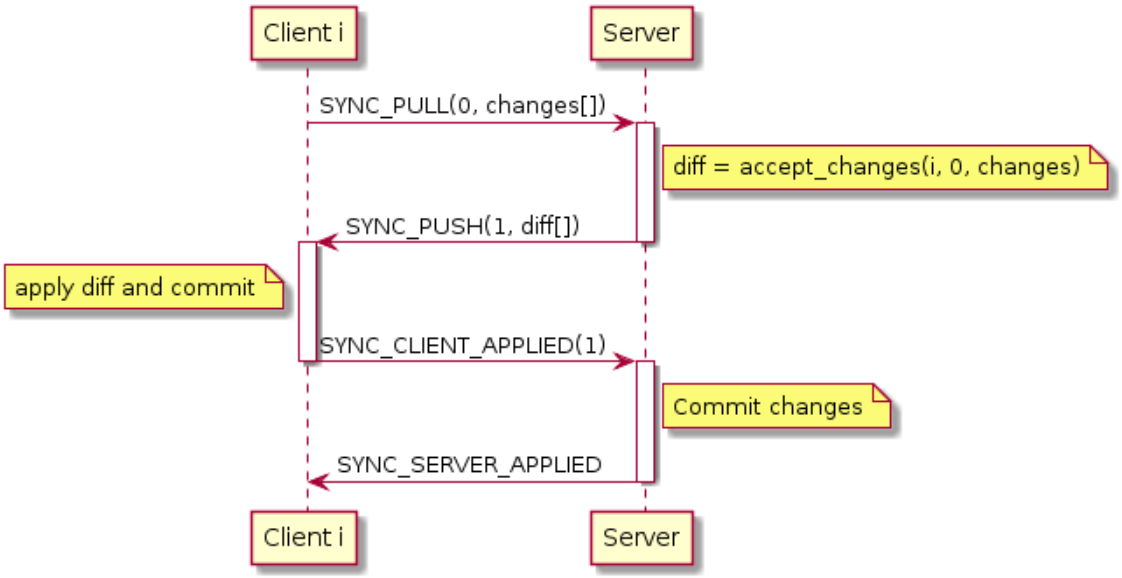

FIGURE 5.2: Synchronization protocol (happy path)

The client database is stored as a stack of commits, similar to the way the server works except commits are totally ordered. Each layer of the stack, called frame, contains

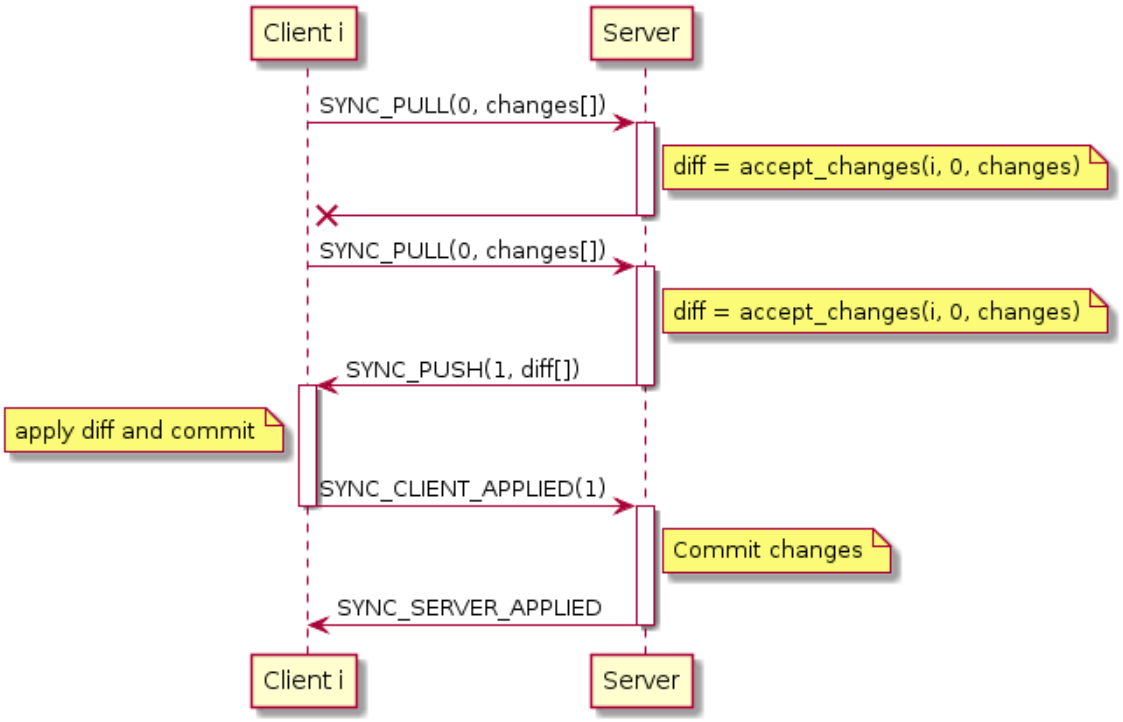

FIGURE 5.3: Synchronization protocol (server diff is lost)

a sequence of changes. It is possible to remove frames and add new ones only from the top of the stack. During normal operation, the topmost frame of the stack contains local changes which have not been sent to the server yet.

The client initiates a synchronization by sending a *SYNC_PULL* message to the server. This message contains the changes from the client's topmost frame. The server processes the changes in the *accept_changes* procedure which returns both a commit id and the diff the client must apply to be in that commit's state. We define this response as the *SYNC_PUSH* message.

```
SYNC_PULL(last_commit_id: number, changes: Diff)
SYNC_PUSH(next_commit_id: number, diff: Diff)
SYNC_CLIENT_APPLIED(commit_id: number)
SYNC_SERVER_APPLIED()
SYNC_ABORT()
```

LISTING 5.1: Synchronization messages with their respective parameters

The client *pops* the topmost frame (effectively rolling back all local changes), creates a new frame with the received commit id and applies the received diff in this frame.

The client is expected to reach a **schema consistent** state from applying the remote diff (since the server has already performed all the checks required to make that state consistent). Therefore the order of the operations in the diff does not matter, and the client skips consistency checks when applying the remote changes.

Then the client acknowledges by sending a *SYNC_CLIENT_APPLIED* message with the commit ID parameter.

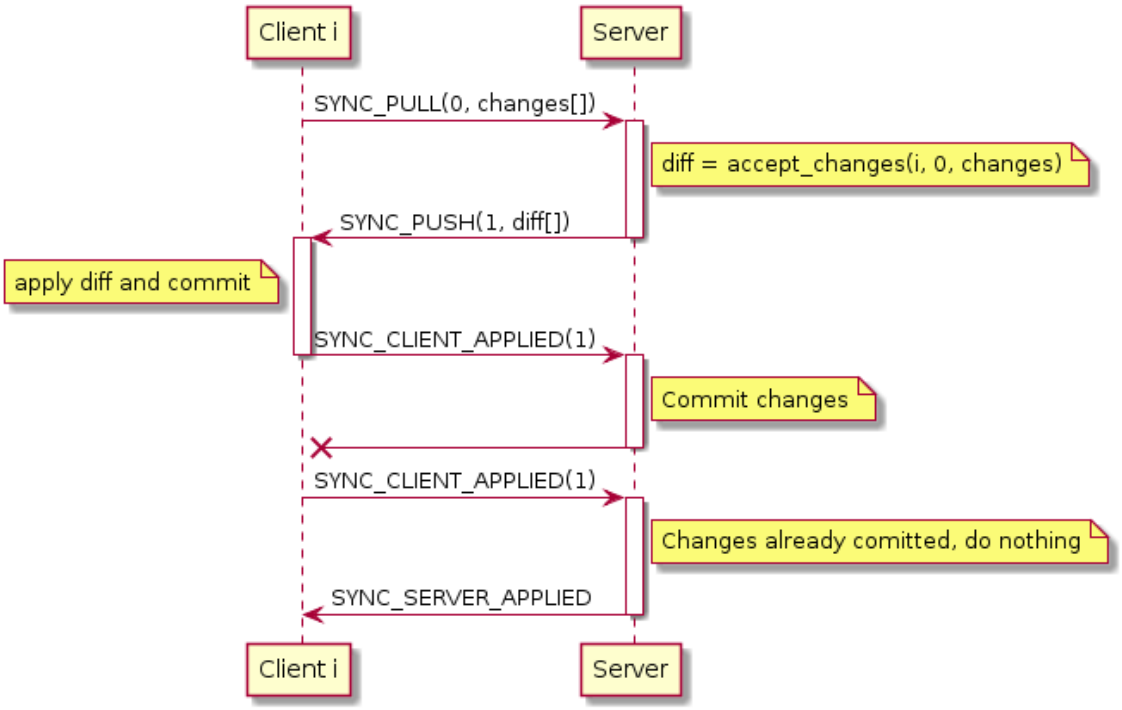

FIGURE 5.4: Synchronization protocol (server acknowledgement is lost)

The server introduces the commit into the database graph with the *accept_commit* procedure and then responds with *SYNC_SERVER_APPLIED* or *SYNC_ABORT* in case of failure.

In order to ensure delivery, the client keeps sending *SYNC_CLIENT_APPLIED* until it either gets *SYNC_SERVER_APPLIED* or *SYNC_ABORT* in response. Note that *accept_commit* is idempotent such that receiving multiple *SYNC_CLIENT_APPLIED* messages is not an issue.

Figures 5.2, 5.3 and 5.4 show how the synchronization protocol behaves with no transmission error, an error during the first exchange or an error during the second exchange respectively. Listing 5.3 defines the procedures used in the protocol.

Listing 5.2 provides the global data types and variables used in the algorithms which are described in listing 5.3.

```
type ClientID: per-client unique identifier
type Commit: Database commit with, a unique "id" scalar attribute
type Queue, Map: Standard data structures.
type [x, y]: tuple of x and y st. [x, y][0] = x and [x, y][1] = y

var pending_heads: Queue<[ClientID, Commit]>
var remote_heads: Map<ClientID, Commit>
var client_tip: Map<ClientID, Commit>
var remote_pending_heads: Map<ClientID, Commit>
var commit_applied: Map<ClientID, boolean>
```

LISTING 5.2: Global types and variables

```
initialization():
    root = new Commit()
    client_tip = (client_id -> root) for each client

accept_changes(clientId, commitId, changes):
    let c = client_tip[clientId]
    let new_head = apply_diff(c, changes)
    remote_pending_heads[clientID] = new_head
    commit_applied[clientID] = false
    return [commitId, diff(new_head, remote_head[c])]

apply_diff(commit, diff):
    let current = commit
    foreach change in diff:
        let conflict = apply_change(current, change)
        if conflict = true:
            current = custom_conflict_resolution(current, change)
    return current

accept_commit(clientId, commitId):
    let head = remote_pending_heads[clientId]
    if head != null && head.id === commitId:
        if commit_applied[clientId] == true:
            return SYNC_SERVER_APPLIED
        pending_heads.push([clientID, head])
        client_tip[clientID] = head
        commit_applied[clientId] = true
        return SYNC_SERVER_APPLIED
    else:
        throw SYNC_ABORT
```

LISTING 5.3: Synchronization procedures

5.5 Reconciliation

The synchronization protocol generates divergent branches which have to be continuously merged in order to reach eventual consistency. We call this process **reconciliation**. Commits are merged using the three-way merging algorithm. In git's implementation of the three-way merge, all files are processed, and then the user is prompted to resolve any conflict. Our version is different in that conflict resolution happens at the heart of the merge procedure, and therefore the state is only partially merged when the user-provided conflict resolution function executes.

```
three_way_merge(a, b):
    let lca = find_lca(a, b)
    if a == lca:
        return b
    if b == lca:
        return a
    let diff = diff(lca, b)
    return apply_diff(a, diff)
```

LISTING 5.4: Three-way merge

The reconciliation procedure runs in the background. It waits until there are at least two active branches in the repository, removes those branches from the active list and asynchronously initiates a merge. The result of the merge procedure is put back into the active list. Remote tips are updated to match this new commit.

```
converge():
    loop:
        wait until pending_heads.length > 1:
            let left = pending_heads.pop_first()
            let right = pending_heads.pop_first()
            do asynchronously {
                let merged = three_way_merge(left[1], right[1])
                client_tip[left[0]] = merged
                client_tip[right[0]] = merged
                pending_heads.push(merged)
            }
        goto loop
```

LISTING 5.5: Convergence function

This function adds one element to the *pending_heads* list for every two elements it removes, as long as there are at least two elements in the list. In the absence of updates, i.e., if no new element is added to the *pending_heads* then that list will eventually contain only one head, with the results of all commits merged into it. The server database is therefore eventually consistent.

Clients which push no update do not generate a new commit but still receive a diff if their remote branch tip changed. Once the database has converged, all clients share the same tip, and therefore all clients will replicate the same state. The system as a whole is therefore eventually consistent.

5.6 Limitations and extensions

5.6.1 Persistence

The initial design focuses on validating a new architecture for data synchronization, and we left durability aside to make it easier to reason about. On the other hand, this choice severely diminishes the practical usability of such system and we ought to look for ways to bring persistence.

One way would be to stream the changes to a RDBMS replicate. We can hook an agent onto the synchronization protocol to get a feed of changes, just like we attempted with CouchDB. Unlike its non-causal homolog, our server is capable of sending the updates in causal order, compatible with **schema consistency**. This means we should not get integrity constraint violations during replication. The synchronization protocol could be extended such that in case the server crashes, synchronization with clients can be restarted from an earlier commit, matching the last commit synchronized in the RDBMS replicate.

In this configuration, the synchronization server acts as an accelerator in front of the database, responsible for handling concurrent updates without blocking and managing conflict resolution. This approach does not solve the problem of synchronizing changes from the RDBMS to the clients.

Another solution is to implement the key-value store on a persistent medium. In this configuration, the synchronization server is the master database itself. An SQL interface could be implemented, for instance as a MySQL adapter. Changes made through this interface would seamlessly replicate to all devices.

Our proof of concept focuses on correctness rather than performance, and therefore the in-memory data structures allow us to exploit the garbage collector already present in the runtime. A persistent solution needs to roll its own garbage collector otherwise it faces the risk to see its disk usage grow unbounded.

5.6.2 Schema definition

The current solution stores schema definitions as code alongside access control rules. One may ask what happens when an updated schema has to be deployed.

On one side bundling together code and data schema can be seen as a benefit to ensure consistency of the application. On the other hand, what happens if the schema changes while the system is running? Transient effects may be observed when data created under some schema is merged according to another schema. The consistency of the database could become compromised.

Instead, the schema could be stored in the database itself. With causal consistency, we can guarantee that updates happening under a later schema revision can no longer be processed with an earlier schema, ensuring that data is always consistent with the latest schema.

This pattern also simplifies deployment of new schemas. The client and the back-end always share the same database schema regardless of the frequency of application updates.

This idea is not an evolution of the current design but rather a different approach with its advantages and weaknesses. It is an interesting path to explore even though it might not provide substantial improvements over the current solution.

5.6.3 Scalability

Our current solution is single-node and single threaded. Moreover, an immutable store can only be updated sequentially and therefore merge operations cannot benefit from multithreading. The big picture looks like, to reach reconciliation, all updates by all

clients would have to have happened in one large sequence spread across the successive merges.

Scaling this system is therefore twofold: how can we horizontally scale the design and how can we overcome the shortcomings of immutable stores concerning parallelism?

The data would need to be distributed across physical partitions to achieve horizontal scalability. Recent work in the domain of distributed databases introduce efficient algorithms to run causally consistent transactions on distributed datasets [4] [23] [24]. One would have to evaluate if those algorithms can be applied or at least adapted to our architecture.

As for parallel access to the immutable store. If the store is distributed across n partitions, it could be possible to run n updates in parallel while maintaining most of the architecture intact. The major challenge is in making sure causally dependent updates do not execute concurrently. An option would be to introduce a scheduler and dependency tracking mechanism to send updates concurrently to different partitions according to their causal relationships.

5.6.4 Performance and Feasibility

Our design favored correctness and simplicity over performance. While we considered asymptotic runtime of operations, actual performance results from many more factors such as data locality, buffers utilization, the frequency of system calls, machine-friendly execution flow, etc. Commercial database vendors have been optimizing their systems for many years and continue to do so. We do not expect a direct implementation of our design to compete with commercial products without some heavy optimization.

We foresee some design decisions as having a significant impact on performance.

Binary search trees are seldom used in database design because they grow deep, and each lookup incurs a lot of sequential I/O with considerable cost of time. N-ary trees such as B+ trees may be more adapted for the task although their implementation is significantly more complex.

Similarly, naive tree implementations have poor data locality which causes cache misses and floods the memory bus, leading to high latency and poor CPU utilization. A popular optimization is to ensure that tree nodes fit well in CPU cache lines and further optimization would aim to keep the topmost nodes in cache as much as possible, speeding up every single lookup. These optimizations are harder to achieve with immutable, and this may be a serious limitation of our design in practice.

While we avoided the topic of garbage collection during this discussion, we do not foresee it as a major performance bottleneck. In fact, many commercial RDBMSs which implement MVCC do have garbage collection but the benefits of MVCC seem to overweigh the garbage collection cost in practice.

Chapter 6

Related work

This section brings some practical context to the theoretical designs seen in chapters 4 and 5 by relating existing technology.

6.1 Cloud solutions

Major cloud vendors offer solutions for data synchronization on mobile in their platform as a service (PaaS) offerings; For instance Google in Firebase[1] and Microsoft with Azure offline data sync[2]. However, they miss a few criteria which makes them unusable in our scenario. For starters, they cannot be deployed on-premise which is a no-go for many business domains. Second, most of them are NoSQL stores which almost certainly makes them incompatible with a SQL backend as we have seen with the case study of CouchDB in section 4.1. The pricing model of Firebase suggests that it is not at all meant for our usage scenario: synchronizing a full business data set across devices would amount to a prohibitively high figure[3].

Finally, these solutions are not open source, and little detail on their internal architecture is given which prevents us from studying those from a research standpoint.

6.2 On-premise SQL synchronization

Software editors offer on-premise synchronization solutions for SQL databases across mobile fleets. We studied SAP MobiLink[4], SAP's proprietary solution for synchronizing databases to mobile clients, which has been used at ELCA in past projects. It turns out that MobiLink works very much like the enhanced SQL-to-SQL design we describe in section 4.2. Some of the problems we encountered and their workarounds are directly discussed in MobiLink's documentation. For instance, administrators can enable a feature to preserve the history of updates and prevent loss of causality[5] as we have seen in section 4.2.2.

[1] `https://firebase.google.com/docs/database/`

[2] `https://docs.microsoft.com/en-us/azure/app-service-mobile/app-service-mobile-offline-data-sync`

[3] Firebase charges 1$/GB downloaded as of writing these lines. In one of our scenarios, it could cost as much as 3000$ a day to beam this data back to devices. Source: `https://firebase.google.com/pricing/`

[4] `https://wiki.scn.sap.com/wiki/display/SQLANY/MobiLink`

[5] `https://help.sap.com/viewer/d2c213f5abb54ffb8c47026a964dd7dc/17.0/en-US/81c670df6ce210149074f2f9dbcc96d5.html`

Microsoft offers a related technology with the Microsoft sync framework. However, due to a documentation of inferior quality, smaller adoption, and somewhat ill-defined scope, we did not study this solution in as much detail as we did with MobiLink. It seems that this solution also follows a similar pattern to that described in section 4.2 with similar pitfalls and issues.

Despite these similarities, there may be differences which we did not spot, for instance, MobiLink may apply optimizations over the encoding of data in transit which we did not discuss. Once again, these solutions are closed-source, proprietary software and we only went as far as the documentation goes to learn about the internals of the systems.

The fact that they are successfully used in commercial products shows that the design is viable in practice despite its theoretical limitations. It does not mean however that integration of these solutions is easy. On the contrary, sources at ELCA reported that the integration of MobiLink has been a painful process and, while it was worth the effort, this experience provided a reasonable basis to start looking for alternatives which were easier to integrate. Besides, software and support licenses for this product are notoriously expensive, often dwarfing hardware acquisition and maintenance costs.

6.3 Graph databases

The so-called *Graph* databases present an alternative to relational databases in the NoSQL world[1]. They expose a model which is close to what is found in SQL databases: documents have foreign key attributes which may point to other documents in the store. Unlike their traditional homolog, graph databases are generally schema-less, i.e., documents have no predefined structure and relationships are defined in the data rather than the model. Graph databases are closer to document stores in that regard.

Popular graph databases include neo4j[2], orientdb[3] and gun.js[4] to name just a few. We did not investigate these solutions in great detail although preliminary research did not reveal suitable solutions for mobile synchronization of entire data sets for offline usage.

As a matter of fact, the solution described in chapter 5 can be seen as an implementation of a replicated graph database. Both offer strong integrity and MVCC, but our solution differs in that it is designed for offline-capable applications and integrates business logic at the core of the conflict handling mechanism.

6.4 CRDTs

Convergent replicated data types (CRDTs, alternatively conflict-free or commutative replicated data types) are high-level data types which ensure eventual consistency provided some basic properties of the underlying system. A wide range of CRDTs has already been discovered to implement counters, sets, and sequences. The fundamental principle

[1] Whether this is a marketing stunt to avoid the negative connotation of *relational* in NoSQL or if it has a real technical meaning is left for the reader to decide.

[2] `https://neo4j.com/`

[3] `https://orientdb.com/`

[4] `https://gun.eco/`

of a CRDT is that the initial value seen at a node may not be the final value, but after a sufficient amount of time has elapsed, all replicates see the same value.

While CRDTs offer useful tools to solve the mobile synchronization issue, they are not complete solutions in themselves. Also if misused, CRDTs can introduce UX artifacts due to the way their value changes until convergence.

6.5 AntidoteDB

AntidoteDB[1] is the reference research platform of the SyncFree european project[2]. It is a geo-distributed database featuring implementations of the Cure [23] and GentleRain [25] protocols. Its main features are: support for Causal+ consistency and CRDTs in a partitioned database replicated over different data centers.

It solves the same technical challenges we faced but in a different context: Antidote is dealing with master-to-master replication across a few data centers whereas we deal with client-server replication across many mobile clients. AntidoteDB achieves high availability under network partitions (AP design in the CAP theorem) but at the same time aims for the highest consistency level achievable under high availability. The consistency guarantees enable support for high-level features such as rich **schema consistency**.

[1] `http://syncfree.github.io/antidote/index.html`
[2] `https://pages.lip6.fr/syncfree/`

Chapter 7

Conclusion

We evaluated several architectures to implement a resilient synchronization channel for offline-capable mobile applications through the implementation of proofs of concept, and we related existing products to those architectures.

In section 7.1 we summarize our findings, highlighting the strength and weaknesses of each solution and we give recommendations to guide potential users to the solution that best fits their requirements. Section 7.2 describes tracks worth investigating to find a solution which can improve over those we present here.

7.1 Summary

We found that the most important criteria is whether the synchronization channel should support relational data. Such support requires **causal consistency** which has implications on the design and scalability of the system. Until recently those consequences were not well-understood [10] and **causal consistency** in distributed databases is still an active research area. Causal systems are much harder to scale-out than their alter ego. This is empirically demonstrated by the fact that most document stores from the NoSQL movement, which claim good horizontal scalability, are non-causal despite the usage benefits brought by causal consistency. The CouchDB synchronization protocol, with many implementations available, is a suitable candidate for building a synchronization channel for non-relational data.

In reality, however, relational data is easier to work with and is ubiquitous in enterprise applications. Each solution we identified to deal with synchronization of relational data had at least one major drawback. Horizontal scalability of the backend database is hard to achieve although it is possible. We found that such database must be designed from the start for the mobile synchronization scenario.

Our attempt at designing a custom storage system for the task proved that a relational database could be implemented on top of a key-value store with some provisions. Our design can distribute the work of resolving conflicts but requires more work to be genuinely horizontally scalable.

A system based on **business transactions** has the most expressive power to model business logic with complex conflict handling. Nonetheless, its programming model can be tricky to understand and can quickly lead to insidious programming errors. This solution may have a significant performance cost due to the amount of custom code which runs on the backend ant the high database isolation level required to avoid anomalies. Furthermore, we could not find a commercial implementation of this architecture.

Architecture (section)	Scalability	Relational	Business Aware	Friendly API
CouchDB (4.1)	horizontal	no	++	+++
Direct SQL (4.2)	vertical	yes	++	++
Business Transactions (4.3)	vertical†	yes	+++	+
Causal consistent (4.4)	vertical?‡	yes	++	+++

TABLE 7.1: Synthetic comparison of the architectures.
† Has contention due to the way RDBMSs implement snapshot isolation.
‡ Design extensions may provide horizontal scalability.

Commercial solutions are available to synchronize existing relational databases with SQL stores embedded in the devices. For instance, SAP MobiLink can be interfaced with most commercial RDBMS. Despite what software vendors might want customers to think, a solution based on this architecture is necessarily a leaky abstraction. We have seen that a naive approach has many problems whose fixes incur various performance hits. Commercial solutions expose controls to enable administrators to trade correctness for speed. These controls define the implicit contract underlying the synchronization channel. Each developer and each operator involved has to agree on and fully understand the contract, otherwise, subtle bugs may arise once production load is applied to the system.

Table 7.1 synthesizes our findings, with references to each section where the reader can learn more about each solution.

7.2 Future work

We have seen that current solutions do not satisfy our requirements entirely. They often expose an error-prone programming API with too many sensitive controls, and most do not scale horizontally. Solutions from the NoSQL world claim horizontal scalability, but they do not provide the guarantees required to support relational data. Poor understanding of the CAP theorem has long provided an excuse and been a hindrance to database authors, preventing them from finding better designs. However, recent work [4] [23] [24] has been exploring the frontier between availability and consistency and opens up a new horizon of scalable and correct relational databases. In our context, this brings an opportunity to offer efficient and easy to use alternatives to the current solutions.

We described an architecture which can challenge current solutions, and we implemented parts of this architecture to verify its feasibility. However, follow-up work is required to find solutions to the remaining problems seen in section 5.6, like disk persistence and partitioning, and to implement a complete proof of concept.

Another approach worth investigating is to take geo-replication protocols, such as Cure [23] or GentleRain [25], and see if they can be adapted to solve our problem. These protocols might not be suited to synchronize a large fleet of devices — some of which may go missing for extended periods of time — which may be a significant issue with such an approach.

Appendix A

Proof of theorem 2.5.1

We can determine all conditions under which a **foreign key** constraint is broken using formal analysis. First, we define a small model for our analysis.

Definition A.0.1.
Let A be the set of all values of attribute a *in table T_1 (referencing table).*
Let B be the set of all primary keys in table T_2 (referenced table).
Let P be the predicate which is true if and only if the foreign key constraint on t *referencing T_2 holds.*

From definition 2.5.1 we get:

$$P(A, B) \equiv \forall a \in A, \exists b \in B : a = b$$

In this model, adding or removing a row in T_1 adds or removes an entry in A, respectively. The same goes for T_2 and B. To simplify the proof, we assume without loss of generality that modifying a row has the same effect as removing and then adding a row.
We can exhaust all conditions by induction. We start from an empty database:

Lemma A.0.1. *In an empty database, all foreign key constraints are respected.*

Proof.

$$P(A, B) \equiv \forall a \in A, \exists b \in B : a = b,$$

with $A = \emptyset$.
This statement is vacuously true regardless of the value of B. □

We are now ready to prove theorem 2.5.1.

Proof.
Given an initial database state A, B such that $P(A, B)$ is true (which we know to be the case of an empty database from lemma A.0.1), we consider all possible evolutions of the state: additions and removal from each set:

An element is added to A: $A' = A \cup a'$

$$\begin{aligned} P(A', B) &\equiv \forall a \in A', \exists b \in B : a = b \\ &\equiv \forall a \in (A \cup a'), \exists b \in B : a = b \end{aligned}$$

$$\equiv P(A,B) \wedge \exists b \in B : a' = b$$

This statement is false if and only if there exist no primary key in B whose value is a′ (1).

An element is removed from A**:** Given $a' \in A$, $A' = A \setminus a'$

$$P(A',B) \equiv \forall a \in A', \exists b \in B : a = b$$
$$\equiv \forall a \in (A \setminus a'), \exists b \in B : a = b$$
$$\equiv \forall a \in A, \exists b \in B : a = b$$
$$\equiv P(A,B)$$

Removal from A may not break the foreign key constraint.

An element is added to B**:** $B' = B \cup b'$
We assume b′ is not already in B which allows us to claim that $\neg\exists a \in A : a = b'$. Indeed, if there was such element it would contradict our assumption that $P(A,B)$ is true. We find that $P(A,B')$ is equivalent to $P(A,B)$ and therefore additions to B may not break the foreign key constraint:

$$P(A,B') \equiv \forall a \in A, \exists b \in B' : a = b$$
$$\equiv \forall a \in A, \exists b \in (B \cup b') : a = b$$
$$\equiv \forall a \in A, a = b' \vee \exists b \in B : a = b$$
$$\equiv \forall a \in A, \exists b \in B : a = b$$
$$\equiv P(A,B)$$

An element is removed from B**:** $B' = B \setminus b', b' \in B$
We assume neither A nor B are empty for $P(A,B)$ is proven to be true when A is empty and we can not remove an element from the empty set.

$$P(A,B') \equiv \forall a \in A, \exists b \in B' : a = b$$
$$\equiv \forall a \in A, \exists b \in (B \setminus b', b' \in B) : a = b$$
$$\equiv \forall a \in A, \exists b \in B : a \neq b' \wedge a = b$$
$$\equiv \forall a \in A, a \neq b' \wedge \exists b \in B : a = b$$

If $\exists a \in A : a = b$ then the statement is false.

Otherwise,

$$P(A,B') \equiv \forall a \in A, \exists b \in B : a = b \equiv P(A,B)$$

which is true by our induction hypothesis.

Therefore, removing an element from B breaks the foreign key if and only if the element appears in A (2).

By induction, we conclude that a foreign key constraint may only be violated if any of conditions (1) or (2) occurs, given an initial database state which does not break the foreign key constraint (such as the empty database for instance).

□

Appendix B

CouchDB and Couchbase Benchmark

Before the implementation of the proof of concept based on the CouchDB synchronization protocol (section 4.1), we benchmarked both CouchDB and Couchbase Sync Gateway to check that they fit our requirements.

B.1 Experimental setup

We created a sample application which synchronizes data across mobile devices using a centralized server and the CouchDB synchronization protocol. Mobile devices were modeled as Node.js applications running PouchDB inside Docker containers. The servers were containerized versions of CouchDB and Sync Gateway limited to 1 logical CPU core and 1GB of RAM.

The application works as follows. Each client periodically creates a *task* document with properties *target*, *payload*, *creation_date*, *completion_date* and *result*. The latter two are initialized empty (*null*), *target* is set at random to the *id* of any client in the fleet (including the emitter) and *creation_date* is set to the current timestamp. The *payload* consists in a simple math problem that the target client must solve. Tasks are created every 500 milliseconds, and the local database is synchronized with the backend every 5 seconds. These numbers were determined empirically to get optimal leverage when varying only the number of clients.

Each client also listens to changes in its local database. Whenever it receives a task whose *target* property matches its own *id*, a client solves the equation and updates the task with *result* set to the solution and *completion_date* set to the current timestamp. The is called *completion time*. Solved tasks are also synchronized with the rest of the database.

The task is expected to have $O(n^2)$ complexity with respect to the number of clients. Within a given time frame, each client sends k updates and receives $(n-1) \cdot k$ updates. Combined, clients generate $n \cdot k$ updates and receive $n \cdot (n-1) \cdot k$ updates which means the server needs to send and receive $n \cdot k + n \cdot (n-1) \cdot k = n \cdot n \cdot k$ individual updates, hence the $O(n^2)$ complexity.

We let the system run for 10 minutes to ensure a steady state is reached; then we measure CPU and RAM usage as well as network traffic at the backend interface and the average task completion time. Completion time is defined as interval between *creation_date* and *completion_date*. It is a measure of the business performance of the system; this measure could be used in practice to define a Service Level Agreement (SLA).

B.2 Results

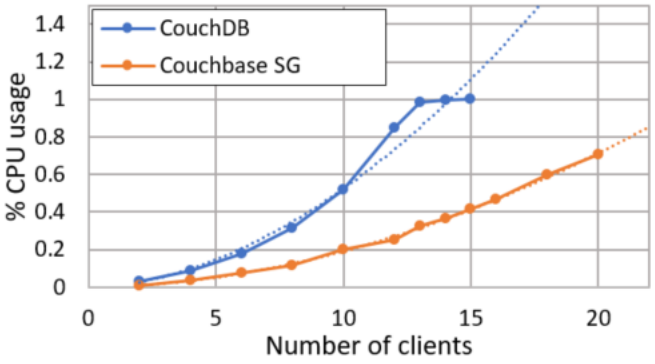

FIGURE B.1: CPU consumption throttled to 1 logical core. The dotted lines show fitted 2nd order polynomials.

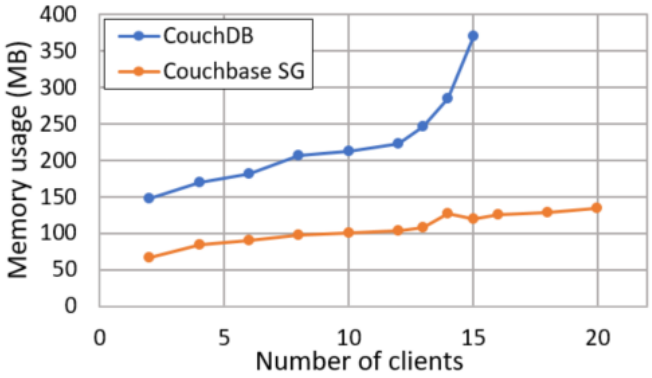

FIGURE B.2: Memory usage. CouchDB's memory spikes as it nears 100% CPU usage. A 30MB measuring error is noticeable.

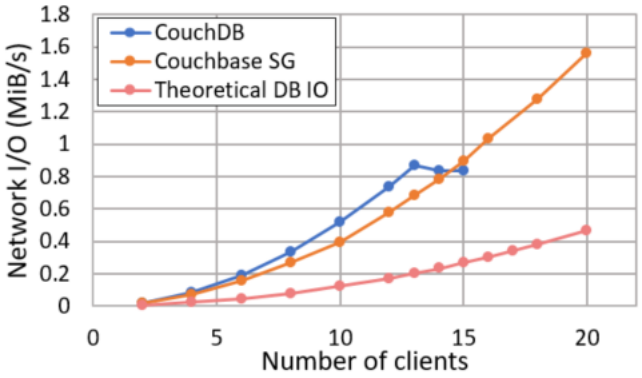

FIGURE B.3: Combined input + output network usage and theoretical baseline. The measuring error is insignificant.

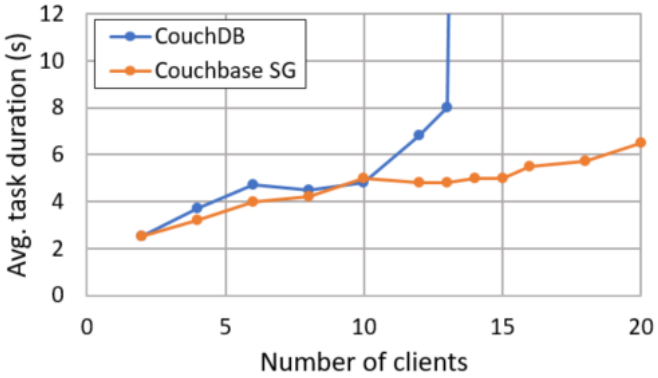

FIGURE B.4: Average task completion time. A client may issue messages to itself which it treats instantly, explaining the trend visible on the left-hand side.

As expected, CPU consumption grows quadratically with respect to the number of clients. Sync Gateway consistently consumes half as much memory as CouchDB, and its CPU usage grows at half the pace of CouchDB. CouchDB shows a breakdown behavior past 13 clients. This corresponds to the moment CPU usage reaches 100%. Bandwidth usage drops slightly and completion time grows unbounded which indicates that the workload is produced faster than CouchDB can consume it. Interestingly, CouchDB's memory spikes as its CPU usage reaches 100%. This is a classic pattern seen when there is not enough CPU left for garbage collection.

At first, Sync Gateway unexpectedly showed a similar breakdown pattern at around 15 clients, although we predicted only 40% CPU usage. We later found that this was because Sync Gateway ran out of cache space. We increased the cache size and, although performance remained the same with less than 15 clients, this allowed Sync Gateway to keep running smoothly with up to 20 clients. We did not perform tests with more than 20 clients because, past this number, our testbench itself was starting to suffer. At that point, we already had the answers we were looking for, so we decided not to pursue pushing Sync Gateway to its limits.

Network usage was similar for CouchDB and Sync Gateway, with a slight advantage for the latter. One would expect to see precisely the same figures as both use the same protocol, but, apparently, they do something a little different. We did not investigate this difference further.

We computed the minimum theoretical bandwidth usage as being the bandwidth required to exchange the raw volume of data created by clients, uncompressed and encoded as UTF-8 JSON strings (which in our case amounts to 290 bytes per task object). Both CouchDB and Couchbase consumed between 3x to 4x this theoretical baseline.

B.3 Conclusion

CouchDB and Couchbase Sync Gateway are both suitable candidates to meet our objectives. Sync Gateway withstood 1.6MiB/s I/O traffic, or 1'600 combined inbound and outbound messages per second. Projections show these figures could have reached up to 2 MiB/s and 2'500 messages, had our testbench not given up before. The performance difference between Sync Gateway and CouchDB can in part be explained by the fact that Sync Gateway uses a dedicated in-memory cache by default. Since the objective of this testbench was not to compare both implementations but rather to see if any would work for us, we did not look at those differences further.

Network usage was considerably higher than our theoretical baseline. The fact that the protocol is based on HTTP, a plain text protocol with a significant header overhead, was foreseen as having an impact on network consumption. An HTTP header section can easily weigh 500 bytes or more, which in our scenario represents 8% of a client's payload. However, the protocol uses multiple requests for each synchronization, so the overhead adds up. Additionally, the payload encoding might be less efficient than what is theoretically achievable, and we did not account for metadata in our estimate.

Nonetheless, the CouchDB protocol favors simplicity and human-readability over performance. Optimizations could dramatically reduce bandwidth usage at the expense of a more complicated protocol.

Bibliography

[1] "Mobile developer population reaches 12m worldwide, expected to top 14m by 2020," *Evans Data Corporation*, Oct 2016.

[2] A. Adya, *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, 1999.

[3] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, pp. 40–44, Jan. 2009.

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

[5] J. Gray *et al.*, "The transaction concept: Virtues and limitations," Citeseer, 1981.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels," *SIGMOD Rec.*, vol. 24, pp. 1–10, May 1995.

[7] P. Bailis, "Linearizability versus serializability." `http://www.bailis.org/blog/linearizability-versus-serializability/`, 2014.

[8] `http://martin.kleppmann.com/2014/11/25/hermitage-testing-the-i-in-acid.html`.

[9] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, pp. 51–59, June 2002.

[10] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 181–192, 2013.

[11] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Scalable atomic visibility with ramp transactions," *ACM Transactions on Database Systems (TODS)*, vol. 41, no. 3, p. 15, 2016.

[12] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, pp. 624–633, Nov. 1976.

[13] A. Burger, V. Kumar, and M. L. Hines, "Performance of multiversion and distributed two-phase locking concurrency control mechanisms in distributed databases," *Information Sciences: an International Journal*, vol. 96, no. 1-2, pp. 129–152, 1997.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS operating systems review*, vol. 41, pp. 205–220, ACM, 2007.

[15] J. Nielsen, *Usability engineering*. Elsevier, 1994.

[16] M. Csikszentmihalyi, *Flow: The Psychology of Optimal Experience*. Harper Perennial Modern Classics, HarperCollins, 2009.

[17] D. Liauw and P. van Hulten, "Demystification of crdt." 2016.

[18] M. Kiefer, "Histo: A protocol for perr-to-peer data synchronization in mobile apps." `https://github.com/mirkokiefer/syncing-thesis/blob/master/syncing-thesis.pdf`, 2013.

[19] S. Newman, *Building Microservices*. O'Reilly Media, 2015.

[20] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 185–196, 2014.

[21] C. Okasaki, *Purely functional data structures*. Cambridge University Press, 1999.

[22] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Conference on the Theory and Application of Cryptographic Techniques*, pp. 369–378, Springer, 1987.

[23] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," in *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pp. 405–414, IEEE, 2016.

[24] D. Didona, K. Spirovska, and W. Zwaenepoel, "The design of wren, a fast and scalable transactional causally consistent geo-replicated key-value store," p. 3, 2017.

[25] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Gentlerain: Cheap and scalable causal consistency with physical clocks," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 1–13, ACM, 2014.