INSTITUTO TECNOLÓGICO DE AERONÁUTICA

Carlos Matheus Barros da Silva

Igor Bragaia

OFFLINE-ONLINE DATA SYNCHRONIZATION FOR DART MOBILE APPLICATIONS

Final Paper

2020

Course of Computer Engineering

CDU 681.3

Carlos Matheus Barros da Silva

Igor Bragaia

OFFLINE-ONLINE DATA SYNCHRONIZATION FOR DART MOBILE APPLICATIONS

Advisor

Prof. Dr. Lourenço Alves Pereira Jr (ITA)

COMPUTER ENGINEERING

São José dos Campos
Instituto Tecnológico de Aeronáutica

2020

A todos aqueles que contribuiram direta ou indiretamente em nossas jornadas até o presente momento em especial às nossas famílias e aos nossos amigos da graduação

Acknowledgments

- Our family for all support on our studies.
- Brazilian Air Force and ITA for the high quality public education in Computer Engineering.
- Prof. Dr. Lourenço Alves Pereira Jr, our advisor, for his support in the studies and development of the project.
- Prof. Dr. Jose Maria Parent for his support and availability to support the studies and development of the project.
- Nacib Neme, sophomore student from 2022' class, who has developed and largely contributed to the development of the first version of the sync protocol applied to the Muskify app in production.
- Our classmates that helped us to complete the course as smoothly as possible even in the tough circumstances.

"If I have seen farther than others,
it is because I stood on the shoulders of giants."
— Sir Isaac Newton

Abstract

It proposes and implements a protocol that allows mobile applications running on distributed clients to get synced to a single source of truth to manage users data in SQL-like databases. This protocol is implemented on a real app developed in Dart using Flutter that communicates to a web server developed in Python. The complete system has been released to the app stores on iOS and Android and the protocol has been tested on the production environment leading to usage metrics of real users as well as tests to ensure the reliability of such technology.

List of Figures

Contents

1 Introduction

This section starts with the motivation and the objective of the work. Besides it presents the work organization which describes the approach adopted to implement the offline-online data synchronization.

1.1 Motivation

Most mobile applications do not require offline-online data syncing because it either works only offline or it works only online by limiting the usage for the case the user has internet connection. On the other hand, we would like to create a mobile application that would let the user keep using the app, even if he has lost his connection, in a reliable way to support offline-online data syncing behind the scene for the user. The main problem of the offline-online data syncing, however, is that it often leads to data syncing conflicts if the same chunk of the database content tries to get modified throughout different applications. Stated this, we tried to develop a syncing protocol that manages client data correctly in order to persist it into a single source of truth by solving syncing conflicts and giving a smooth experience to the client user. We also state that this problem is often faced in modern distributed software such as email providers or file sharers such as One-Drive, Google Drive or Dropbox. Many companies have patented their own technology but there is not plenty of resources and open-source packages available on trending mobile development technologies such as Dart, recently released by Google.

1.2 Objective

We propose a protocol and implement it in Dart that allows our app to have the offline-online data syncing feature. This implementation is based on principles described on patented technologies but aims to solve the specific problem where we sync SQL-like data across Dart applications and a remote database accessible through a Python RESTful API.

Our primary intention was to build an app capable of storing our customers' data in remote servers, thus they would be able to use different devices signed in using the same account, in such a way that all of those devices would remain in sync with the same source of truth.

The problem of the aforesaid idea is that, in a real situation, the mobile connectivity of our user is, in fact, intermittent. Therefore creating a system that relies on the premise of a continuous syncing with the server is, in practice, not feasible. Either the user will not be able to use the app in a connection instability situation, or he will use and get errors or lose data in such circumstances. Thereupon we had to figure a way to not rely on stable connection and at the same time maintain the requirement of having a server that stores the user state and allows multiple devices signed in with the same account.

The goal became to provide a user experience where our mobile application customer was able to seamlessly change from an online to an offline state with no impact on his utilization whatsoever, that is, he wouldn't even notice he potentially was offline for a moment. We would like to develop, thereby, a mobile application that would offer both offline and online usage, in such a way that the goal is to not impact the user experience in the event the user loses his internet connection.

To accomplish the aforementioned, the app might have both local and remote data management in order to not be dependent on the connectivity. Furthermore, once the app acquires access to the internet connection, it might perform the data syncing between the local data and the remote data. This way, connectivity is not a requirement but is a desired feature because it allows the user to get his data synced across the cloud and different devices. In this scenario, merging client data from single devices into a single source of truth leads to possible syncing conflicts that might be correctly addressed to come up with consistent data.

1.3 Work Organization

In the next sections, we will go through related works, mentioning american patents that aim to solve similar problems but don't exactly solve it for a mobile application by offering an open-source package in the Dart language. We will also discuss in-depth the motivation of this system and then describe the system design and architecture as well the details of the sync protocol and how it works. Finally we will go through the results and the learned lessons of releasing an app to iOS and Android using this developed technology and following user feedback and database usage.

2 Related work

Some existing works relate to this one on the field of creating an online and offline mobile app data synchronization infrastructure.

On the David P. Hil's Patent (HILL, 2011), Microsoft Corporation, we can see a description of a synchronization system that accomplishes most of what we need, it is a distributed system that supports multiple clients connected to a server that can handle conflict from those clients. The clients support both offline and online optimization.

The problem with David P. Hil's work is that it lacks an objective description of the communication protocol since we aimed to develop the solution, the mere superficial description of the operation is not enough. Therefore that work is useful as a form to understand more about this kind of synchronization technology, although it does not solve our problem.

On the Sriram Srinivasan's Patent (SRINIVASAN *et al.*, 2011), Microsoft Corporation, describes a File System capable of synchronizing changes made on a directory while offline, that is, with no internet connection.

That was also a good information resource for our basement, but it doesn't solve our problem. Srinivasan's work is focused on file system synchronization on a personal computer, it has a specific description of procedures related to File Systems that are not what we intend to build or work with. It also lacks an objective description of the communication protocol used.

On the Patent (BLOCH *et al.*, 2007), Eric D. Bloch, Laszlo Systems, describes a system where an application would have a local cache in such a way that while online the client would fetch the server and the data response in the application would be stored on a local cache. On the possibility of the app getting offline, the application would fetch the data from the local cache. This description was useful since it includes many details of the communication logic via flow charts.

Although it lacks an open source implementation or library that we could use in our project. Therefore we still have to develop our own system. We ended up making a very different system with significant changes on some communication logic on our work to

make it more useful for our use, that is a mobile application synchronizing data with a server.

On Patent (VELLINE *et al.*, 2010) from Sybase Inc., the inventors described a protocol that is used to sync different data types across different platforms. This protocol converts the data objects to a common data format, wherein said common data format differs from said native formats, that is stored on a common place and retrieved on future sync requests. On our project, however, we would like to sync only SQL-like data, which means that both the client and the server might store models as tables that are defined exactly on the same way to constrain the scope of the solution.

On the Patent (SMITH, 2017) from Box, INC., the inventor Michael Smith describes a patented technology to offline access and synchronization that uses message passing through push notifications on mobile devices but does not implement or release public packages for the Dart programming language or the Flutter framework released by Google, which is our main goal on this project.

Finally, on the Patent (SMITH *et al.*, 2011) from Microsoft Corporation, the inventors have patented the syncing technology used on softwares that required syncing as Outlook(®). Although its principles and high-level protocol definition have inspired our own protocol implementation, Microsoft does not offer an open-source implementation of such patented technology that can be used throughout other mobile applications.

3 Bachelor thesis proposal

The offline-online data syncing proposed in this paper relies on principles and hypotheses that can lead to an implementation using any technology stack available for mobile applications and web servers development.

3.1 System design

The mobile application system design consists of an app, a client that runs on mobile devices, a web server that runs on the cloud as well as persistent and volatile storage on both the client and the server.

Our system is divided in four tiers with the following responsibilities:

- Tier 1: there are multiple mobile devices in which one has access to its local database. If there is no internet connection, data is persisted on the local storage only. Otherwise the client tries to sync it.
- Tier 2: there is the web server that is deployed on the cloud. The requests from the mobile devices are load-balanced into different workers that run distributed and independently.
- Tier 3: there is a memory cache with light input-output bandwidth. It is responsible for providing frequently requested data such as authentication tokens.
- Tier 4: there is the database with heavy input-output bandwidth. Reading data from disk takes heavy computational resources and might be avoided whenever it is possible by using the cache tier.

Given this proposed architecture, we decided to implement this using the following technologies:

- Tier 1: the mobile application is developed using the language Dart and the framework Flutter. We have chosen Flutter because the app is written in Dart but directly

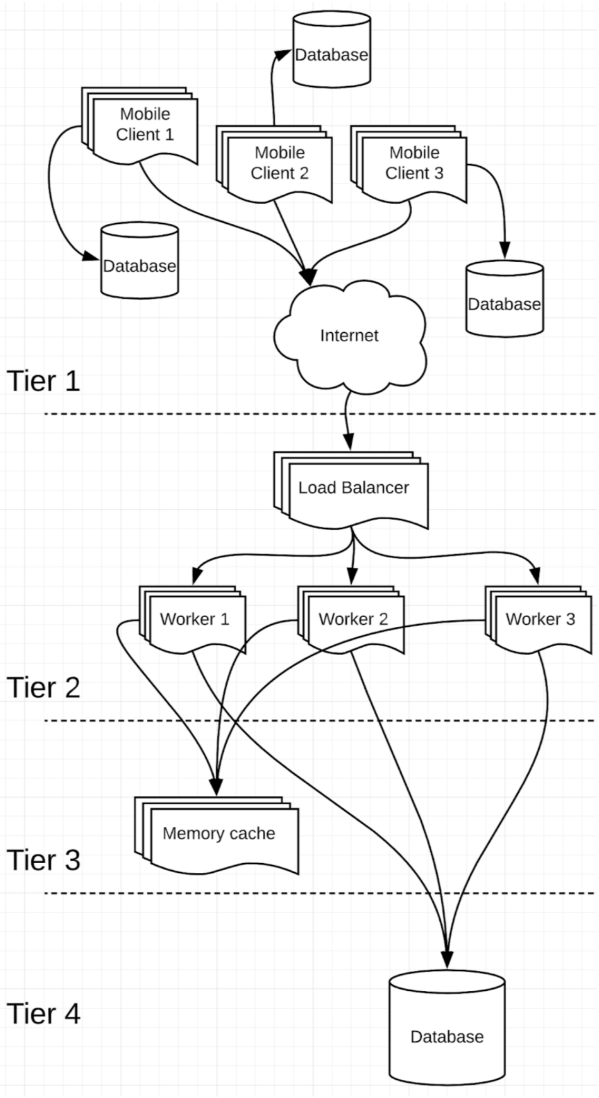

FIGURE 3.1 – System design for a mobile application that supports offline-online data syncing

built into the native mobile language (Objective-C, Swift, Kotlin, or Java) and is not above some sort of interface that can lead to eventual processing lag. The mobile application also has a local database for offline storage that is implemented using a SQLite database. The SQL queries are written using the PostgreSQL language and there is a Dart package to allow the client to connect directly to its local database. The app also uses key-value based shared preferences to store light user information.

- Tier 2: there is a web server that is developed using the language Python and the framework Flask. This web server is a RESTful API that processes client requests that aim mainly to handle authentication (it supports Facebook, Google, and Apple auth services) and data syncing between the client and the remote single source of truth.

- Tier 3: there is the memory cache implementation using Redis.

- Tier 4: there is a remote database implementation using Postgres.

Tier 2, 3, and 4 are deployed over a cloud platform as a service. Such platforms are interesting because they simplify the DevOps work by offering easy horizontal and vertical scalability. It also offers a load-balancing infrastructure that can distribute requests based on application-specific data. The load-balancing then ensures the reliability and availability of the service by monitoring the health of the applications once it only sends requests to servers and applications that are able to respond in a timely manner. On the other hand, the trade-off of using a cloud platform is that you recurrently pay for a DevOps job that you delegate to the platform.

In our case, we have decided to use the Heroku infrastructure as a cloud platform as a service due to the worldwide popularity and reputation of this service. We plan to migrate to Amazon Web Services in the future when the complete system is stable, having more control of the cloud infrastructure and spending less money, but for now Heroku is a good and practical use.

3.2 Syncing protocol

The syncing protocol defines a way for syncing data between multiple clients and a server in a reliable way. It guarantees the offline-online app usage on mobile applications.

The implemented protocol consists of requests made by the client to a server in a specific sequence aiming to let the user be not affected by the connection loss. The syncing also occurs when the user signs in to let the user be always synced for the first use on a new device.

Thereupon, the protocol has model objects that can be synced by some specific message type such as a create, update or delete message which are logged as diff structures. The syncing between a server and a client happens through a series of sync process requests that have diff structures as payloads which might follow specific requirements to be accepted and applied to the previously defined models.

3.2.1 Objects

Objects are the most fundamental aspect of a model that can be synced. Each object has a set of typed fields and the `uuid` field is mandatory for all objects and it's assigned during instantiating (`CREATE` message).

In an SQLite-like database, objects are equivalent to tables. The client and server must agree on existing object types and their structure before sync occurs. Field types

may be: `int`, `string` (fixed or variable size), `float`, and `list`'s of those types. Nested lists (lists of lists) and references to other objects are not supported at this point.

3.2.2 Messages

The sync protocol defines a sort of messages that can be performed to the data being synced. They are the `CREATE`, `UPDATE` and `DELETE` messages.

3.2.2.1 Message types

- `CREATE(char objtype[40], char uuid[128], data)`: This message instantiates an object, assigning it a UUID, and its initial data. It is equivalent to the SQL statement `INSERT INTO objtype (uuid, ...data...)`
- `UPDATE(char objtype[40], char uuid[128], char field_name[128], data)`: This message updates fields of a stored data for an object instance with given UUID. It is equivalent to the SQL statement `UPDATE objtype SET fieldname=...data... WHERE uuid=uuid`
- `DELETE(char objtype[40], char uuid[128])`: This message deletes an instance with given UUID. It is equivalent to the SQL statement `DELETE FROM objtype WHERE uuid=uuid`

3.2.3 Diff structure

On a syncing process, every database message is logged as a diff. All clients keep track of local changes by storing local diffs and submits it to the server when syncing. The server keeps the most recent diff sequence properly merged.

```
1  struct diff {
2    int upstream_height;
3    int timestamp;
4    int optype;
5    union {
6      struct {
7        char objtype[40];
8        char uuid[128];
9        char* data;
10     } opdata_create;
11     struct {
12       char objtype[40];
13       char uuid[128];
14     } opdata_delete;
15     struct {
16       char objtype[40];
17       char uuid[128];
```

```
18        char field_name[128];
19        char* data;
20      } opdata_update;
21    };
22  }
```

Code 3.1 – Description of the diff structure

The parameters description is

- `upstream height`: This the height all clients and the server agree on for that diff.
- `timestamp`: UNIX timestamp of when the message happened.
- `optype`: A constant (int) meaning either `CREATE` (0), `UPDATE` (1) or `DELETE` (2)
- `objtype`: object type name
- `field_name`: field name
- `uuid`: object UUID
- `data`: serialized object or field data

The `upstream_height` is important to create a logical clock in the distributed system and order diffs when merging them. To understand the relationship between `upstream_height` and `timestamp`, consider two diffs `A` and `B`. Note that `A.upstream_height > B.upstream_height` does not imply `A.timestamp > B.timestamp` and vice-versa. It can happen on a practical example as following

- Client 1 creates object `aaa` and syncs `(timestamp 1, upstream height 1)`
- Client 2 downloads sync from server
- Client 2 deletes object `(timestamp 2, no upstream height yet)`
- Client 1 updates `aaa` and syncs `(timestamp 3, upstream height 2)`
- Client 2 syncs (diff with timestamp 2 gets upstream height 3)

3.2.3.1 Sync process

In order to perform a sync, the client reaches the server with a `SYNC_BEGIN` request. The server replies with a `SYNC_BEGIN_RESP`. The client then receives all necessary diffs and merges them with the pending diffs. Finally, the client submits the merged diffs with a `SYNC_CONTINUE` request. The server may either accept or reject it. The process ends with a `SYNC_CONTINUE_RESP` message from the server.

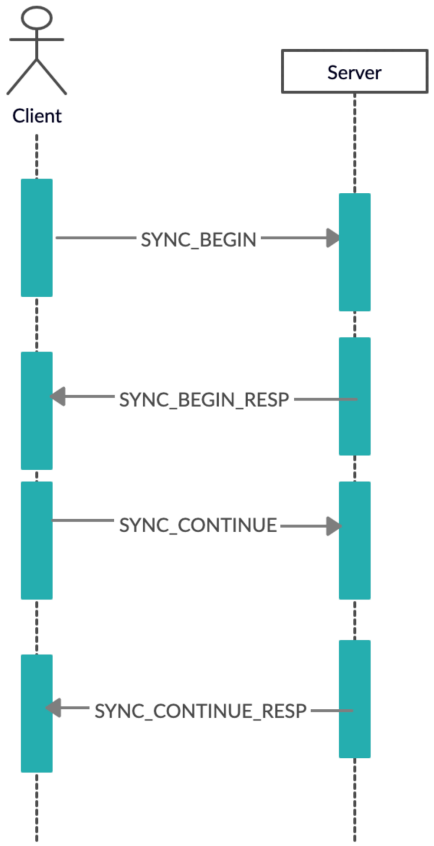

FIGURE 3.2 – Sequence diagram for the sync process requests.

Also, note that these messages are meant to be implemented on top of HTTP or any protocol.

3.2.3.2 Sync request and response messages

- `SYNC_BEGIN(int last_sync_height)`
  - `last_sync_height`: is the last known `upstream_height` for this client.
- `SYNC_BEGIN_RESP(int status, int current_height, data)`
  - `status`: `0` if the sync process may continue or `code` otherwise. Failure reasons include:
    - `ERROR_SERVER_BUSY` (1): server is busy performing another sync for this user
    - `ERROR_INVALID_HEIGHT` (2): client sent an invalid `last_sync_height`
  - `current_height`: The highest height known by the server at that point.
  - `data`: diffs with `upstream_height` in the range `(last_sync_height, current_height]`.

- `SYNC_CONTINUE(data)`
  - `data`: merged diffs with correct `upstream_height` to be included in the server's log.
- `SYNC_CONTINUE_RESP(int status)`
  - `status`: `0` if the sync process succeeded or `code` otherwise. Failure reasons include:
    - `ERROR_INVALID_DATA` (1): client sent invalid diff data that could not be unserialized
    - `ERROR_NO_SUCH_OBJECT` (2): client tried to update or delete an object that the server doesn't know
    - `ERROR_OBJECT_EXISTS` (3): client tried to CREATE an object with an uuid that already exists
    - `ERROR_MALFORMED_UPDATE` (4): client sent a malformed UPDATE diff
    - `ERROR_MALFORMED_UPDATE_UUID` (5): client tried to change an object UUID via UPDATE
    - `ERROR_INVALID_SYNC_TOKEN` (6): implementation specific

3.2.3.3 Merging diffs

After the client receives a `SYNC_CONTINUE_RESP` message and a list of diffs from the server, it merges the downloaded diffs (hereby referenced as `S`) with the local diffs using the algorithm described below. Let `U` be the list of diffs that are going to be sent upstream.

A description of each step of the sync protocol merging diffs procedure flowchart can be found on the Appendix A.

3.3 Preliminary results

3.3.1 Automated tests

In furtherance of ascertaining the implementation correctness of the sync system, we developed an assemblage of unit tests that cover most of the sync protocol specifications. Those unit tests were conceived both on the app-side and the server-side sync.

Following numerous iterations of improvements on the sync protocol implementation, and after the development of supplementary tests to include most of the aspects of the sync descriptions, the system operated as coveted. From that point, we, therefore, started the tests with some actual people utilizing the app.

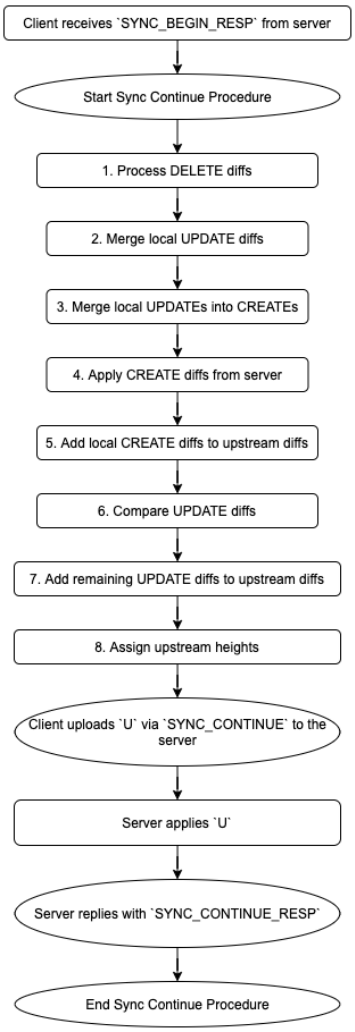

FIGURE 3.3 – Overview of the flowchart sync protocol merging diffs procedure description.

3.3.1.1 Metrics

Alongside the app release for Android and iOS in both test and production environments, we have been collecting users data in order to measure the feasibility of the proposed solution. In terms of metrics, we can observe that the number of diffs stored on the Diff table by the syncing protocol on the Postgres database on Heroku have grown more than four times the amount of the model data itself stored on its model table.

This way, it is evident that this proposed syncing process can lead to an overuse of computational resources and can be optimized on future works.

This database growth over weeks also reveals mobile application-specific insights. On the developed application, for example, for each week there is a peak on Monday and a valley on Saturday, which means that people start the week planning their routines but the app usage decreases along the week. Also, by 08th June, 2020, the app was used only by testers for both iOS and Android but on the mentioned date it has been released for

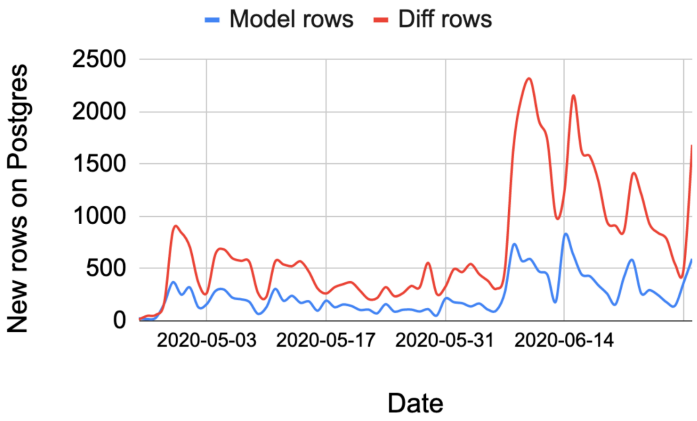

FIGURE 3.4 – Storage results for a specific model object synced often during app usage. It compares the model table rows that are synced and the diff rows also stored on the server throughout the syncing process.

Android, considerably increasing overall synced data. Finally, on 29th June, 2020, it has been released for iOS, what also indicates an usage increase for the near future.

3.3.2 Lessons learned

Subsequently to building the sync system and setting it to operate with all the automated tests, we finally released the system alongside the application for internal tests. Initially, those tests contemplated 6 people for 2 days, then we extended for an external test with roughly 300 people for a duration of approximately 2 months.

During that test period, no problem associated with the sync was reported, all users were able to use the app online and offline and through multiple devices. The sync was working as expected.

After the test phase, we published the app to the public. With more than 4 thousands of downloads and hundreds of active subscribed users, the app sync attests itself as a trustworthy solution for our primary app-server connection problem.

3.3.3 Mobile app release experience

The presented project has been developed under the raised necessities alongside the development of the `Muskify` app. It is available for Android users on the PlayStore, and for iOS users on the App Store. For more information about the app itself, go to www.muskify.app.

3.3.4 Conclusion

The functional implementation of a sync system was achieved from the foreknown development, as stated, such description resolves a nonexclusive sync problem that can be accomplished amidst any stack of technology. The utilization in the production environment of the technology by thousands of people indicates a reliable performance. The aforesaid, alongside with the assortment of the automated test allowed us to esteem the sync procedure as a stable operation.

This way, the sync protocol proposed on this paper and implemented on a real mobile application released to iOS and Android has been based on the documentation of american patents that implemented technologies similar to the aforementioned. Although it has been applied to apps written using the framework Flutter in the language Dart, it could be implemented in any technological stack and applied to other different contexts of software that requires offline-online data sync other than those apps.

Finally, we see that the described protocol needs up to four times extra storage than the data we really need to sync on both client and server on a frequently used mobile app. It reveals a possible leakage on the proposed protocol that leads to the overuse of computational resources that can be explored and optimized on future works.

3.4 Next efforts

3.4.1 Proposal

As our bachelor thesis, we aim to develop an open-source Dart package that will encapsulate the proposed offline-online data synchronization. It will allow developers to use this technology in other different mobile applications.

This package may be able to deal with failure cases in distributed system such as a scenario where

- the single source of truth goes down and the system has to recover
- a distributed and independent client loses connection on during a sequence of sync requests leading to a stale or corrupted state

The package may also be published and documented as a public Dart package contributing to the open-source community.

3.4.2 Experiments

We may accomplish unit and integration tests that will test the protocol dealing with possible failure cases making sure the correctness. It may also have mathematical demonstrations of how the system avoids deadlocks, for example.

We may also improve the necessary bandwidth and extra memory necessary to store data due to the sync protocol other than the data the clients wants to sync in fact. This metric, i.e., how many computational resources are used over time for the proposed protocol will be a key result to also develop a light protocol with the least downside possible for a third party user that wants to implement data sync on his mobile applications.

3.4.3 Schedule

- July: Related works bibliography reading covering patented technologies and distributed systems specific knowledge.
- August: Development of the protocol, extending the already proposed protocol based on the patents and on the bibliography.
- September: Protocol implementation and Dart package setup.
- October: Protocol tests and measures. Package release. Finish paper.
- November: Presentation and final paper delivery.

Bibliography

BLOCH, E. D.; CARLSON, M. D.; KANG, P.; KIMM, C.; STEELE, O. W.; TEMKIN, D. T. **Enabling online and offline operation**. [S.l.]: Google Patents, set. 25 2007. US Patent 7,275,105.

HILL, D. P. **Framework for managing client application data in offline and online environments**. [S.l.]: Google Patents, ago. 9 2011. US Patent 7,996,493.

SMITH, E.; STILLION, C.; ASH, A. **Offline synchronization capability for client application**. [S.l.]: Google Patents, jun. 21 2011. US Patent 7,966,426.

SMITH, M. **Mobile platform file and folder selection functionalities for offline access and synchronization**. [S.l.]: Google Patents, set. 26 2017. US Patent 9,773,051.

SRINIVASAN, S.; JOLLY, T. E.; KRUSE, D. M.; AUST, B. S. **Synchronizing for directory changes performed while offline**. [S.l.]: Google Patents, nov. 22 2011. US Patent 8,065,381.

VELLINE, A.; KACIN, M.; SODHI, R. S. **System, method, and computer program product for online and offline interactive applications on mobile devices**. [S.l.]: Google Patents, out. 19 2010. US Patent 7,818,365.

Appendix A - Synchronization Algorithms Flow Charts

A.1 Flowcharts describing the Sync Continue phase

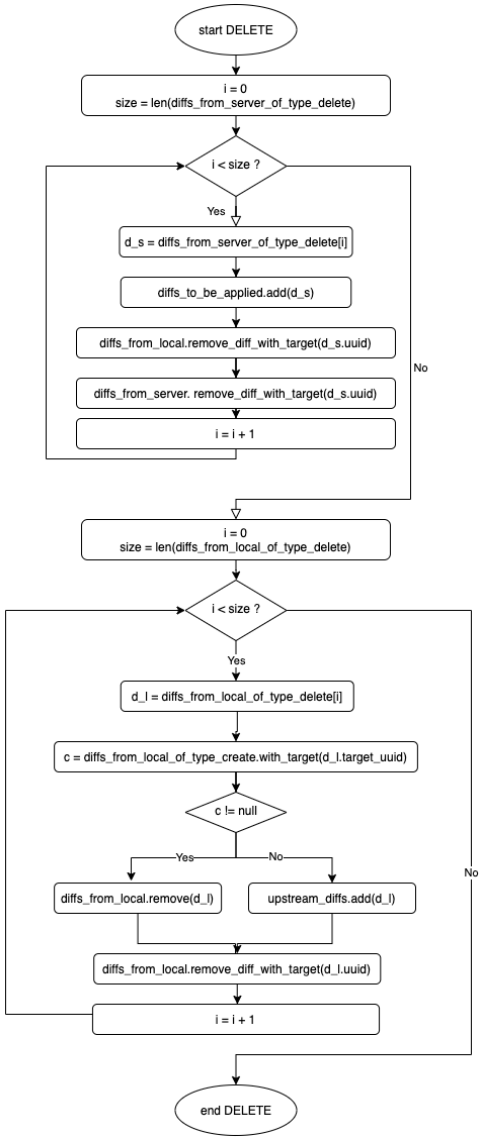

FIGURE A.1 – Flowchart describing step 1: Process DELETE diffs.

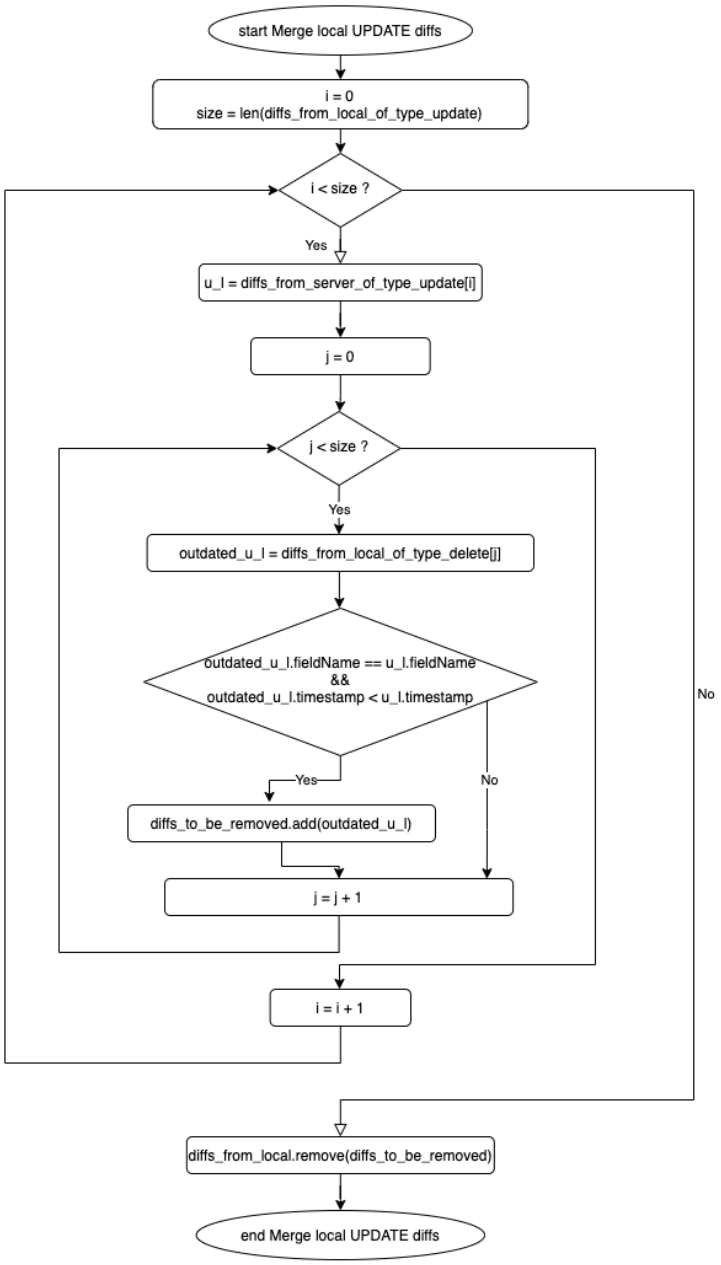

FIGURE A.2 – Flowchart describing step 2: Merge local UPDATE diffs.

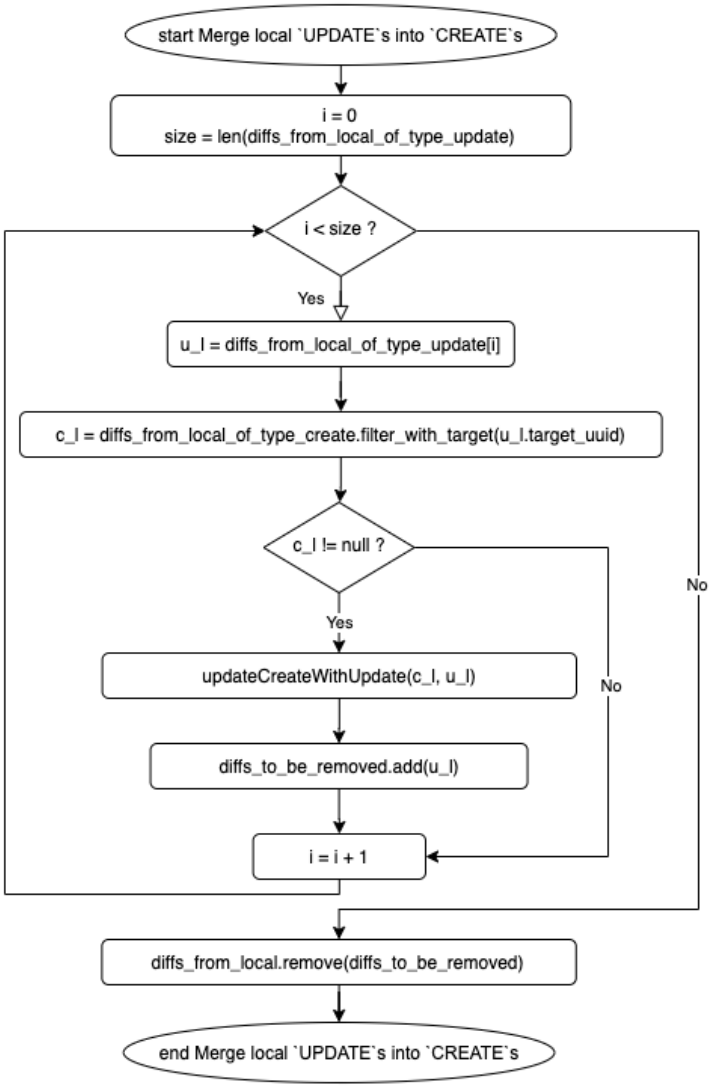

FIGURE A.3 – Flowchart describing step 3: Merge local UPDATEs into CREATEs.

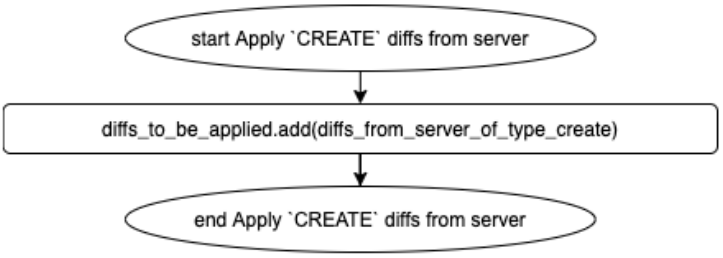

FIGURE A.4 – Flowchart describing step 4: Apply CREATE diffs from server.

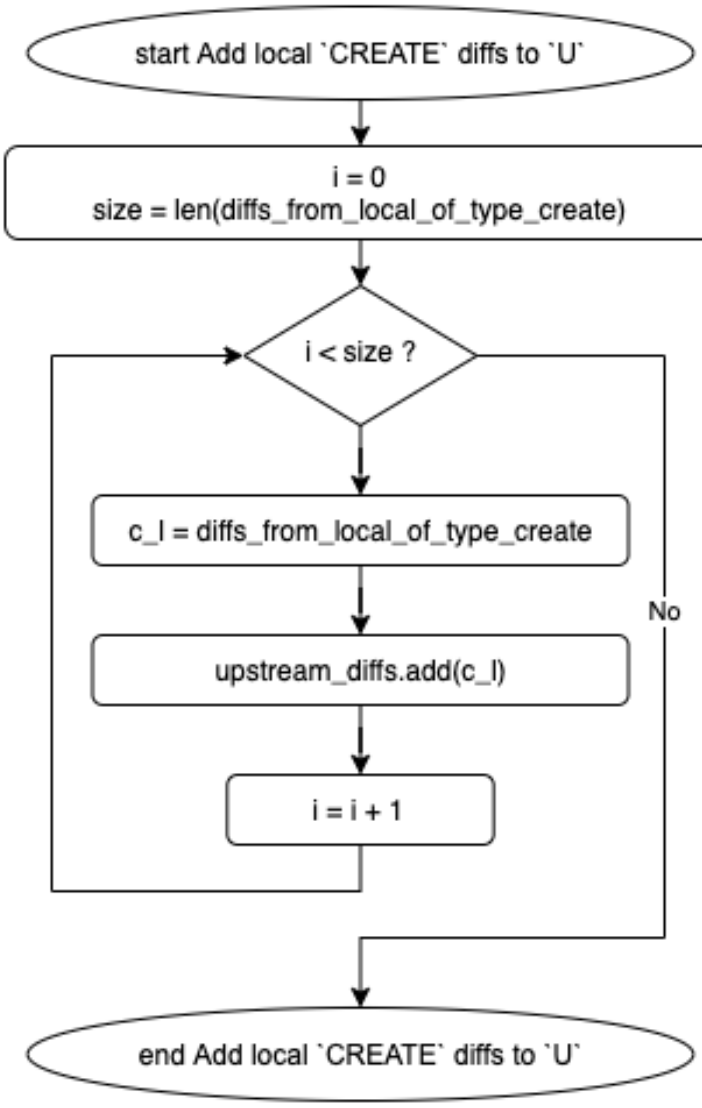

FIGURE A.5 – Flowchart describing step 5: Add local CREATE diffs to upstream diffs.

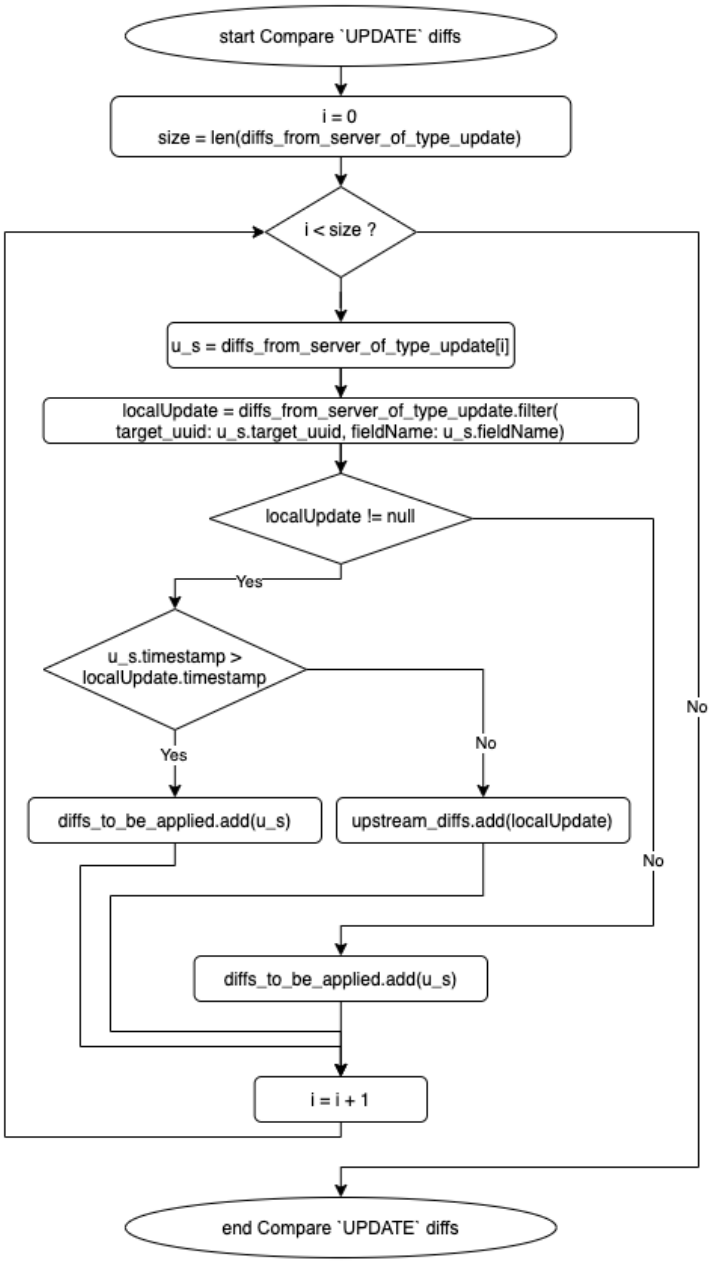

FIGURE A.6 – Flowchart describing step 6: Compare UPDATE diffs.

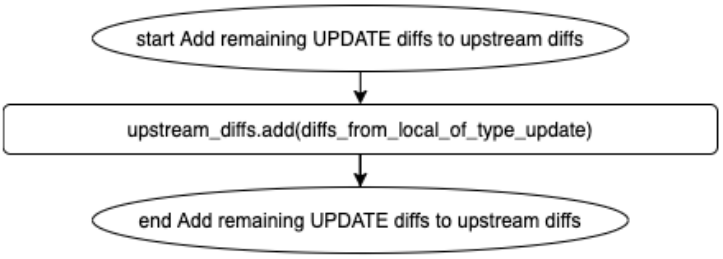

FIGURE A.7 – Flowchart describing step 7: Add remaining UPDATE diffs to upstream diffs.

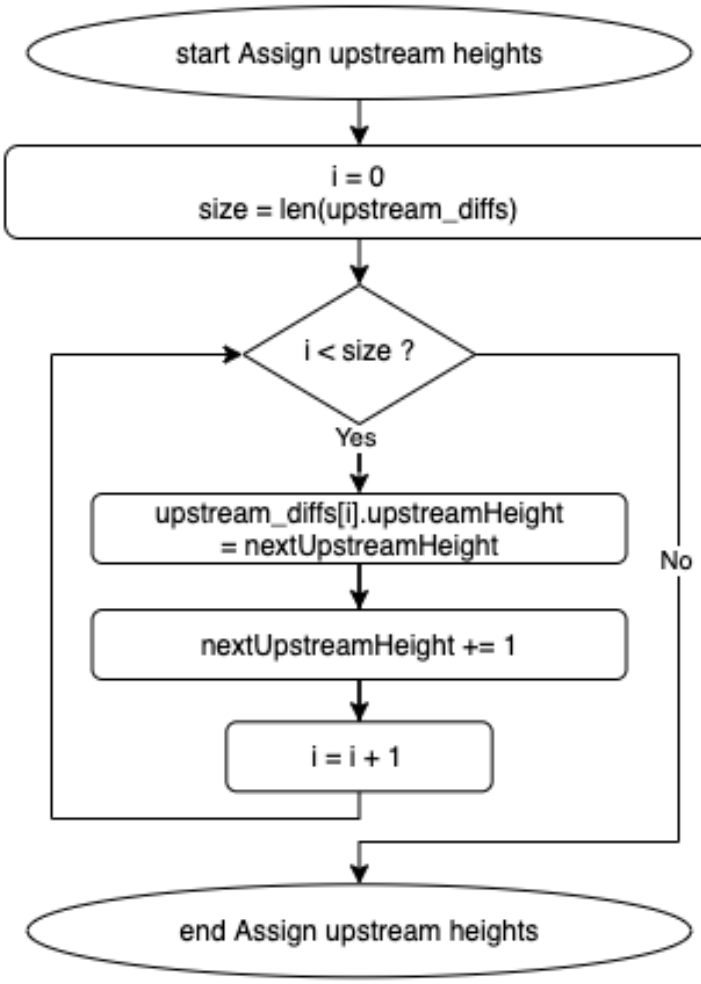

FIGURE A.8 – Flowchart describing step 8: Assign upstream heights.

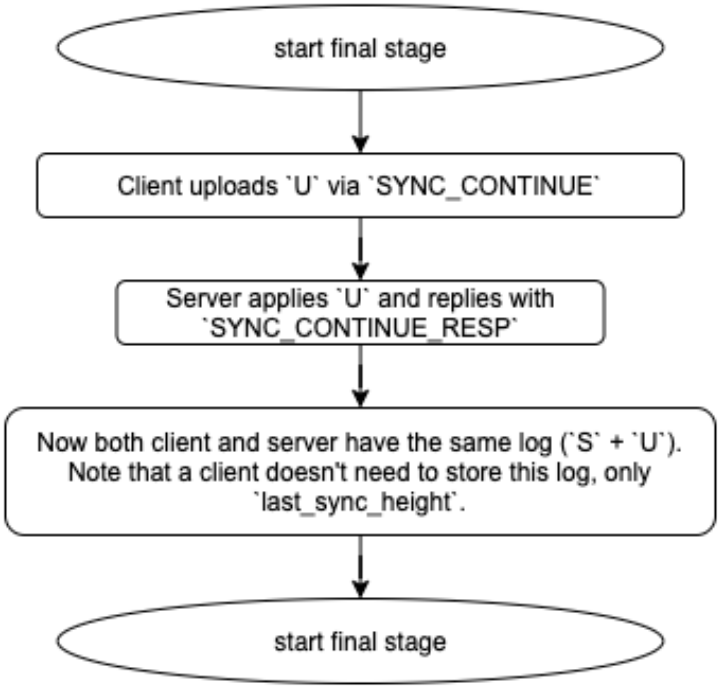

FIGURE A.9 – Flowchart describing the producedure final stage.