Individual Diploma Thesis

IMPLEMENTATION OF A CLIENT - SERVER SYNCHRONIZATION ALGORITHM FOR AN ONLINE PET INFORMATION MANAGING SERVICE

Georgios Moullotos

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

MAY 2017

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Implementation of a Client - Server Synchronization Algorithm for an Online Pet Information Managing Service

Georgios Moullotos

Supervisor
Chryssis Georgiou

The Individual Diploma Thesis was submitted for partially meeting the requirements of obtaining the degree of Computer Science of the Department of Computer Science of the University of Cyprus

Acknowledgments

Many people are to thank for their assistance and role in completing this thesis. Firstly, I would like to express my gratefulness to my research supervisor, Associate Professor Chryssis Georgiou, for granting me the great opportunity of working with him. His guidance, support and contribution were really important to conclude this thesis. I would also like to thank my friends and fellow students from my undergraduate years for being there for me whenever I needed them and for the countless hours we spent studying and discussing our subjects. Also, I would like to thank my personal friends that were there for me to cheer me up when I needed it the most. Moreover, I would like to express my admiration and gratitude to the founders, Kyriakos Stavrou and Demos Pavlou, and to all the team of the 11Pets Startup Company for their huge assistance in completing this work and for the priceless knowledge they have shared with me. Last, but of course not least, I would like to thank my family for believing in me, bearing me and being the greatest of all supports anyone could wish for.

Abstract

The Online Pet Information Managing Service is basically one of the numerous services that the Startup Company 11Pets has to offer. The main service that the company offers, is a platform consisting of several interconnected clients (web services and mobile apps) which allows its users, among others, to maintain information regarding pets and their wellbeing. This information is stored on the local database that each device that owns the application maintains. For the simple reason that each user owns more than one devices, the ideal scenario for a system is to be able to provide a way to maintain these different databases consistent between them. This offers the user the ability to switch between the devices owned and carry out the same tasks without any differences. Of course, to provide such features a set of issues have to be considered regarding the environment of the system. Some of these particularities are:

- Mobile environment provides no guaranties regarding communication with servers and requests.
- Such features should run on background of mobile applications and not allocate too much resources.
- Most cost should be put on device's and not on server for scaling.
- A clean solution, from a software engineering perspective is needed, as implementation goes on iOS as well.

The study and implementation of an algorithm that will satisfy the above particularities and will provide the synchronization of data amongst devices, as well as the study of the general problem of synchronizing data is the subject of this Individual Diploma Thesis.

TABLE OF CONTENTS

Chapter 1 1
1.1 Motivation 1
1.2 Purpose 2
1.3 Methodology 3
1.4 Contributions 4
1.5 Thesis Organization 5
Chapter 2 6
2.1 Introduction 6
2.2 System Architecture 7
2.3 Unique Sequence Number (USN) 8
2.4 Conflict Resolution 9
2.5 Conclusion 10
Chapter 3 12
3.1 The Synchronization Algorithm 12
Chapter 4 20
4.1 Introduction 20
4.2 Required Structures 20
4.3 Unit Tests 24
Chapter 5 26
5.1 Introduction 26
5.2 Apache to Volley 26
5.3 Special Cases 29
5.4 Multiuser Scheme 31
5.5 Implementing the Synchronization Algorithm 33
5.6 Conflict Resolution Mechanism 38
5.7 Debugging 41
Chapter 6 46
6.1 Introduction 46
6.2 System Usage 46
Chapter 7 57
7.1 Metrics Overview 57
7.1 Metrics Evaluation 58
Chapter 8 61
8.1 Summary 61
8.2 Future Work 62
Bibliography 64

LIST OF FIGURES

Figure 1 - System Architecture 8
Figure 2 - Synchronization Scenario 9
Figure 3 - Conflict Resolution Scenario 10
Figure 4 - Synchronization Cycle 13
Figure 5 - Referenced Entities 14
Figure 6 - Process Server Changes - Entry Deleted on Server 15
Figure 7 - Process Server Changes - Entry not Deleted on Server 17
Figure 8 - Process Device Changes 18
Figure 9 - Client - Server Communication 21
Figure 10 - Deletion Side Effects 22
Figure 11 - Apache 27
Figure 12 - Volley 27
Figure 13 - Upload Files 28
Figure 14 - Download Files 29
Figure 15 - Special Case - Seeded Maintenance Types 31
Figure 16 - Multiuser Scheme 32
Figure 17 - Phase A - Setup 34
Figure 18 - Phase B - Process Server Changes 35
Figure 19 - Phase C - Process Device Changes 36
Figure 20 - Phase D - Process Files 37
Figure 21 - Conflict Resolution Interface 41
Figure 22 - Add Pet 47
Figure 23 - Pet Gallery 48
Figure 24 - Pet Schedule 48
Figure 25 - Pet Daily Care 49
Figure 26 - Pet Medical 49
Figure 27 - Adopt 50
Figure 28 - Professionals 51
Figure 29 - Register 52
Figure 30 - Restore 53
Figure 31 - Synchronize 53
Figure 32 - Setting Up 54
Figure 33 - Processing Data 54
Figure 34 - Downloading Files 55
Figure 35 - Uploading Files 55
Figure 36 - Conflict Resolving 56
Figure 37 - Synchronizing an empty Database for the first time 58
Figure 38 - Synchronizing without any changes 59
Figure 39 - Synchronizing with one new entry 59
Figure 40 - Synchronizing with one updated entry 60

Chapter 1

Introduction

1.1 Motivation 1
1.2 Purpose 2
1.3 Methodology 3
1.4 Contributions 4
1.5 Thesis Organization 5

1.1 Motivation

In the modern era, cloud computing and data are thriving. The need for synchronization is essential in order to provide consistency amongst data. Synchronized data give the advantage to their owner to manage them from any device that has the system's application installed. However, keeping the data stored only on the cloud is a bad practice and can lead into many unwanted situations. First of all, what it is done when an internet connection is not available? What happens if the server falls? The application can simply do nothing as all of the data is unreachable. That is why, there is a need to keep data stored locally as well.

The 11Pets is a pet-centric company that offers a platform consisting of several interconnected clients that help its users to maintain information regarding pets and their wellbeing [13]. These platforms basically are an android application, an iOS application and a web application. The number of the users of the 11Pets android application is about eighty thousand (80,000). Day by day this number is increasing rapidly and the piece of information that comes up with these users becomes more and more valuable. What happens now when a family, that are responsible for taking care the same pet, want to have the application on their own device but have the same piece of information among each other? Imagine that one member of this family feeds the pet and informs the application about this event. How can the rest of the family be informed so they will not proceed, on their turn, to feed the pet? This is the problem that the synchronization algorithm of this thesis is called to solve. These actions and data will be propagated amongst devices sharing the same accounts and help people

coordinate in order to take care of their beloved pets. Thus, the algorithm will not only benefit the users with the advantages we have seen but will also allow the company to gather up all of this important information. The vision of 11Pets is to unite all pet-interested parties and create a large ecosystem on where they will communicate with each other. These interested parties can be professionals in the field of pets, like vets and trainers, pet shelters and rescue centers as well as each one of us that love and care for pets.

Each individual system must be in place to provide some guarantees based on the environment of the system in question [3]. The environment of our system consists of the following particularities that need special attention:

- Regarding the mobile environment:
  - No guaranties are given about the communication with the server.
  - Build-in classes and their methods are evolving rapidly.
  - Speed is essential.
  - Must be lightweight. Features should run on background and not allocate too much resources.
- Our application's database scheme is evolving as new features are implemented.
- Most cost should be put on devices and not on server for scaling.
- A clean solution, from a software engineering perspective is needed, as implementation goes on iOS as well.

With this in mind, there is a need for a mechanism that will synchronize all of the devices that want to share their data and keep them consistent between each other.

1.2 Purpose

In order to achieve all that was described so far, we need to put together a wide range of techniques regarding the computing field, some of which we will see in the chapters to come. *Synchronization* is the capability of maintaining the same piece of information alongside all devices sharing the same account and the main server [4]. At this point, it is worth mentioning that a main body of the algorithm studied on this Individual Diploma Thesis preexisted on paper as a draft, designed by members of the 11Pets Company. We did a lot of modifications in order to fit it to our system's needs and we proposed and discussed a lot of changes and improvements to its basic structure. So basically, what we are asked to do is, starting from the initial scheme of the database, to implement a mechanism that would synchronize the data with all of the particularities our system has. Our goal, therefore, is to design and implement a synchronization algorithm that will solve the data sharing problem

we have studied earlier and meet all of our expectations. The implementation of this algorithm is the subject of this thesis.

1.3 Methodology

To achieve our final goal, which is the implementation of the synchronization algorithm, we need to recognize and organize all of the tasks that must be done. The tasks presented below, concern both the implementation of the algorithm as well as various structures and mechanisms that are essential for the algorithm to be functional. The methodology way we followed is:

- General study of the problem: the synchronization problem as well as the conflict resolution problem and similar concepts were studied.
- Study and understanding of the algorithm: as mentioned before the algorithm existed in a primitive form on the paper. In depth study and understanding of the algorithm was essential in order to become confident enough to start and complete its implementation.
- Implementation of required structures, on the application and on the main server, that will allow sending and receiving data from both of these sides to each other: we have two main structures that needed implementation, the DTOs (Data Transfer Object) and the DAOs (Data Access Object).
- Study and implementation of Unit Test to check the correctness of the conversion from DAO to DTO and vice versa: the unit tests are a special piece of the whole application development cycle and the way of their designing and implementing was studied before implemented.
- Conversion of android application requests to the server from apache protocol to volley: For a better communication with the server, both for prior existing requests of the application and for the new requests that came with the implementation of the synchronization algorithm, we converted all of the communication from apache to volley. The implementation of such requests was done with the use of asynchronous tasks.
- Handling special cases: apart from the usual tables that were included in the synchronization cycle there were some tables that needed special handling. Automatically generated entries must be created in a deterministic way and seeded data should be the same for all devices.

- Implementation of the algorithm: the existing drawings on the paper were transformed into working code using the supportive structures we talked about on earlier bullets.
- Implementation of the conflict resolution mechanism: this mechanism is able to dynamically detect conflicts and provide three options, discussed on a following chapter, in order to solve them.
- Debugging of the algorithm: numerous tests on local and live versions were done in order to prepare the algorithm for its final releases.
- Empirical Evaluation: in this thesis, we scratched the surface of some performance metrics of the algorithm. These are time complexity of some essential operations of the algorithm. In future stage a more analytic study and evaluation of the algorithm will be done to proceed to optimizations as well as a study to find the best moments on a usage of the app to start the synchronization algorithm.

1.4 Contributions

Summarizing all of our efforts in this Individual Diploma Thesis the main and most important contribution is the deployment of the synchronization algorithm on the devices of the Company's users. We are very pleased that we were able to reach the end of the line and provide the users with something more practical and efficient than what they were accustomed to use. In short, here is a list of what we have accomplished:

1. We took an algorithm on paper and studied it.
2. We implemented it and while doing so we:
    a. Found cases that the database's schema and assumptions of the algorithm did not match.
    b. Found types of data that needed different handling (automatically generated entries and seeded tables).
    c. Found conflict resolution problems.
    d. Run different tests and methods of debugging.
    e. Data sanitization (clean up user's databases, e.g. delete zombie entries)

    We proposed a set of solutions for each individual case and picked the most appropriate to solve it.
3. Implemented a multiuser scheme.
4. Released the algorithm for public use as the main feature of the 11Pets android application for data backup and retrieval.

1.5 Thesis Organization

So far, we have seen a summary of the various aspects that compose this Individual Diploma Thesis. In the first chapter we began, by mentioning the reason this idea was born and the need for its implementation. Afterwards, we set our main goal and our expectations that must be reached until the end of this study. Later on, we focused on our methodology, describing with more details what we have to implement from the very beginning to the end of this Individual Diploma Thesis. We then separated the tasks in steps that have a logical sequence. Finally, we studied our final results and evaluated ourselves and the work we have done on this long journey of studying and coding.

So, in the chapters to come, we will be referring more specifically in the technical parts and see how all of these combine together and compose the desired synchronization algorithm. In the next chapter, we will study some general problems in the field of the synchronization, conflict resolution and more. These studies will give us the desired knowledge and influence so that we will be able to solve each of the issues we will have to face. On the third chapter, we will study the implementation of our algorithm in depth by dividing our implementation in separate pieces and analyzing both the algorithmic way of thinking as well as the code itself. The fourth chapter will provide a manual of how the algorithm behaves in the real world. Moreover, we will evaluate our algorithm and results on the fifth chapter and finally, we are going to close this Individual Diploma Thesis with our conclusions and future work on the sixth and last chapter of our work.

Chapter 2

Background and Related Work

2.1 Introduction	6
2.2 System's Architecture	7
2.3 Unique Sequence Number (USN)	8
2.4 Conflict Resolution	9
2.5 Conclusion	10

2.1 Introduction

Synchronization refers to the idea that multiple client devices, sharing the same account, are to link or handshake so as to reach an agreement depending on a set of data [1]. Synchronization is a general problem that has been studied in the computer science field for many years. It all started with the need of low level processes to communicate with each other in order to accomplish a specific task. The need of synchronization has moved towards higher levels and now it finds appliance in all of the application's layers. Synchronization concerns both the communication between processes as well as the consistency of data. This Individual Diploma Thesis focuses on the second aspect of synchronization, the data synchronization. To achieve the data synchronization, a lot of patterns are known to exist [11]. Each one of these synchronization patters is applying to some specific system's needs and comes with its benefits and liabilities. Synchronization is used by many known applications and services to provide users the best experience in storing their data. Some of this applications that use their own synchronization algorithms are Dropbox, iCloud, Google Drive and many more [12]. Studying all these examples, we are called to combine them and design our own synchronization algorithm that best fits our systems architecture and environment.

2.2 System Architecture

By having a main server and a group of clients that communicate with the server, there are many ways to achieve storage and synchronization of data. Firstly, a fully online implementation may exist. This means that all data is stored only on the main server and all devices must communicate with the server directly to fetch the needed data. Hence, working offline is not possible. Another way to implement the synchronization is the fully offline way. In this feature, each client keeps all of the data locally on its device. There is no communication with the server and the data are not replicated or shared at all. Lastly, there is the intermediate way of data storing and synchronization, which is the one we will use for our system. In each synchronization execution, the data of the server and the device are synchronized and kept updated on both remote and local devices.

As previously mentioned, the algorithm of this thesis will implement the *intermediate* way of data storage. What a system basically gains from this method is both the ability to work from more than one device as well as the capacity to work offline. Working offline is essential for the 11Pets application because it maintains essential medical records that may need to be accessed at any time. In order to implement this algorithm, we must study numerous concepts and techniques that will allow this dynamic synchronization between the main server and the local devices. Some of these techniques that are essential for our implementation presented in the following sections.

An important characteristic of our system are the data conflicts. *Conflict* is the state of the system where two or more replicas are found to have a piece of information that is different, whilst it is considered as the most recent piece of information on all replicas [6]. In this case, the system is called to be able to resolve it and restore balance with the correct piece of information to dominate all others.

On our system, synchronization was achieved on the granularity of a database. This meant that one is to send and receive the whole database as a file in order to backup and restore it, respectively. This way doesn't allow any resolution between conflicts because the one file overwrites the other and it is hard to locate specific differences. What our algorithm brings, is the granularity synchronization. Synchronization will be done separately for each table and each record of the database. This is the harder of the two to implement, but it gives the opportunity to handle inconsistencies dynamically and resolve them.

Our system consists of a main server and lot of client devices. The system's job is to answer all of the clients' requests and synchronize and store their data. Each device, on its side, has a local database store on its memory. The database of our application consists of numerous tables that are hierarchically structured. Meaning that there are references from one

table to the others and these connections synthesize together a tree. Our algorithm is responsible to go through all of these tables and with the *record granularity* synchronization to keep data consistent. In Figure 1 we depict a simple outline of our system architecture.

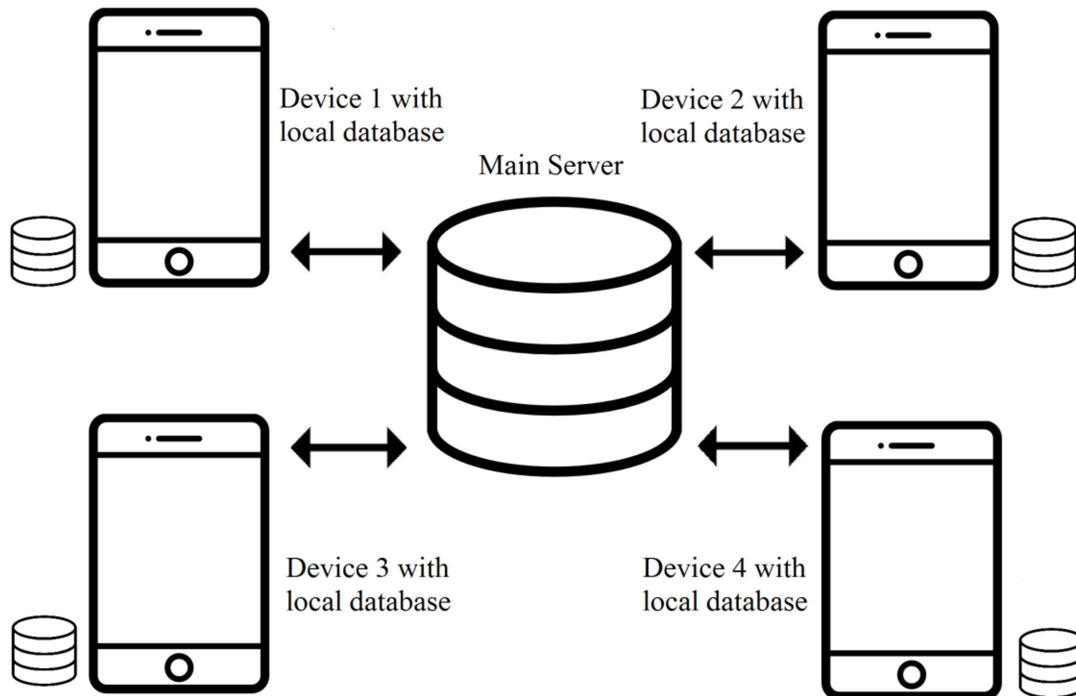


Figure 1 - System Architecture

2.3 Unique Sequence Number (USN)

The synchronization algorithm targets at synchronizing the database of the granularity of a record. For this specific reason, we added the Unique Sequence Number field [5]. The USN is kept per record and shows the current version of this entry. The USN is only increased by the server. The server keeps track of the greater USN of each record and is responsible for updating this field. When a record is modified on the server or a device sends an update of the record to the server, the USN of this record is updated and more specifically it is increased. Next time another device wants to see the changes its database has with the main server, it is not hard to track whether there are different USNs between the local record of the device and the record stored on the main server. This is the most common way to track differences between records on two different machines and this is the technique that we will be using on our data. Such scenario is showed in Figure 2. On the left, we can see the server having a newer version of the entry than the device has. After the synchronization, the two entries are now the same. On the other hand, we have the same USN but with the entry being

dirty on the device. When synchronized, the entry will be updated on the server and the USN will be increased on both the server and the device.

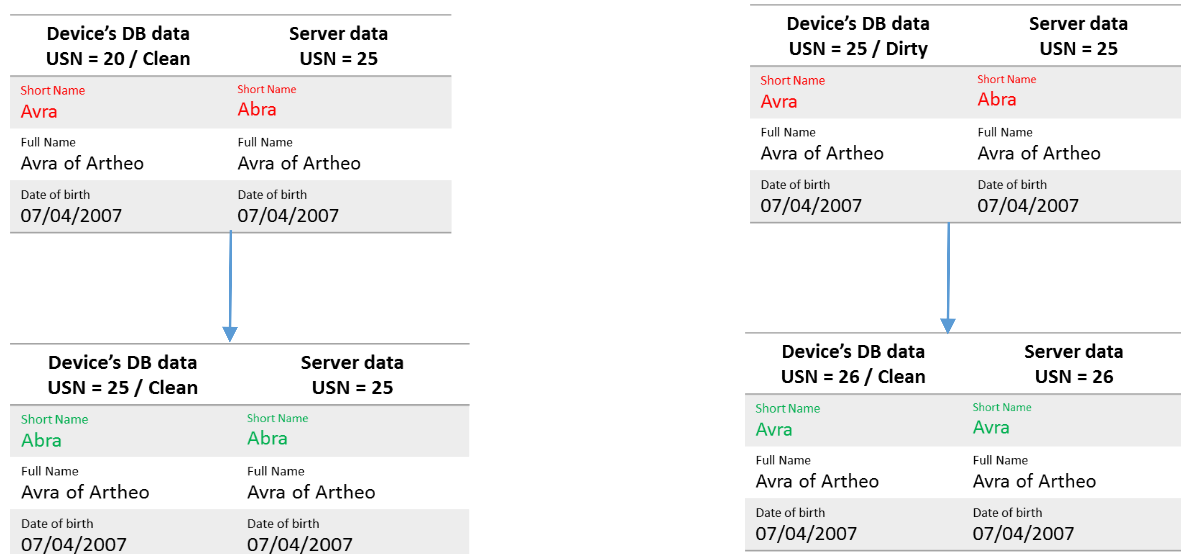


Figure 2 - Synchronization Scenario

2.4 Conflict Resolution

In our system, a conflict occurs when two or more devices, starting from the same piece of information, end up having different data while having the same version of information. For example, consider the following scenario: Assume that the main server keeps a record with USN of 25 and two devices that have their local copies updated and their USN of the specific record set to 25. Now, both of these devices change this record on their local devices. The first one to achieve synchronization with the main server will sent the updated entry to the server and get the new increased USN. What happens when the second device starts to synchronize with the server? The USN has changed on the server but so does the record on the local device. Which is the correct piece of information?

There are many ways to resolve conflicts with each of them giving different participation of the user. The most common way to resolve a conflict is to give the user the following three options:

- Keep the remote piece of information, namely the data held on the main server.
- Keep the local piece of information, namely the local data of the device.
- Compare the timestamps of the two records and keep the most recent.

As you may see these three ways of conflict resolution give the user the absolute privilege to keep its desired piece of information. There are systems that may only provide two of the three above options or even choose one of them without even asking the user about its preference. We shall follow the above guideline and give our users the comfort to resolve their conflicts the way they want. In Figure 3 that follows, we can see a scenario of a conflict and its resolution. Two devices are synchronized with the server and thus have the same data. Both modify their corresponding entries but the first device synchronizes first and sends its update to the server. When the second device wants to synchronize a conflict is spotted. The user chooses that the correct piece of information is its own and when the first device synchronizes again the consistency of the data is restored.

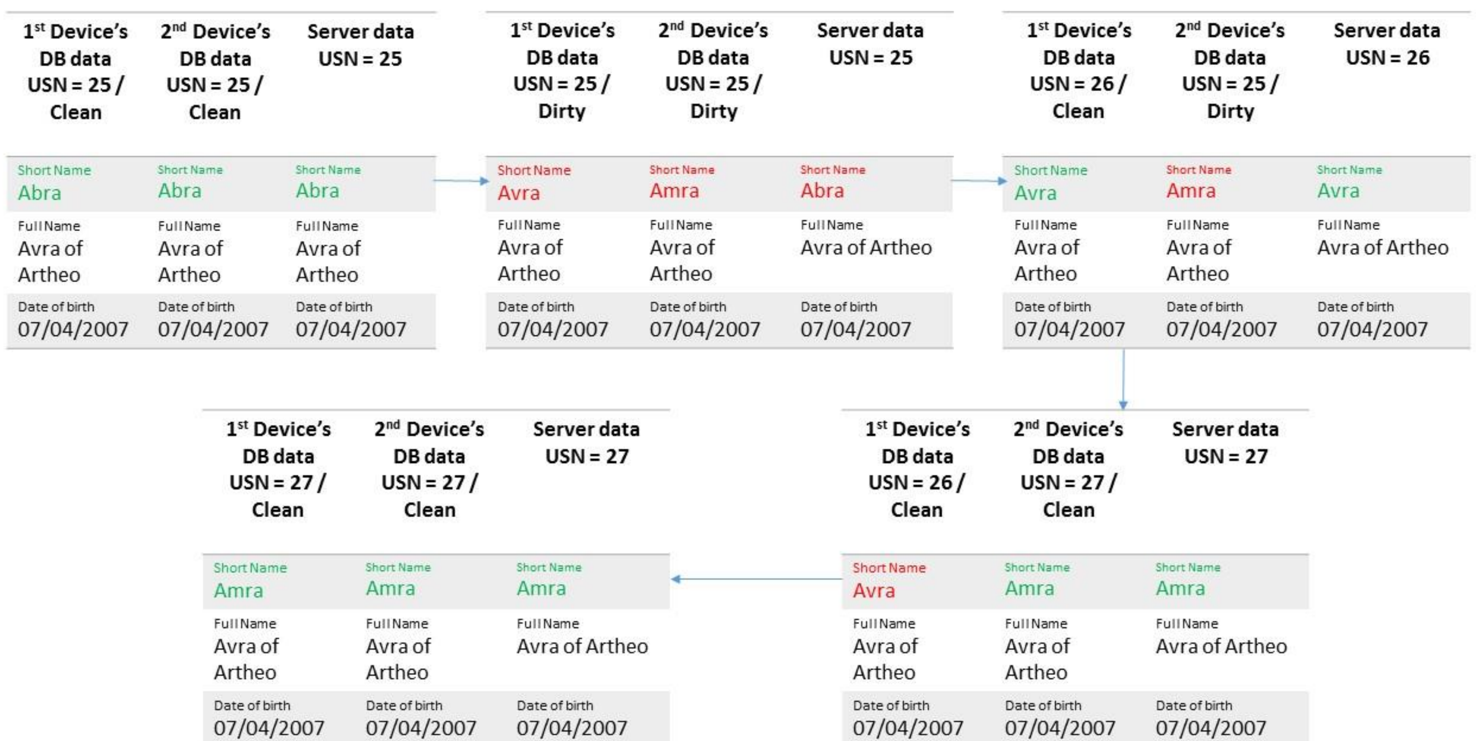


Figure 3 - Conflict Resolution Scenario

2.5 Conclusion

Having all of the above general aspects well studied we come in place to see how all of these apply to our problem. Our system consists of the server which contains the most updated copies of our data. Each user, on the other hand, stores data on any number of desired devices with each one having its own local replica. A numerous of particularities

depending our system must be considered. These particularities are the ones described in the first chapter. Moreover, the communication and the initiation of the synchronization process will be done by using a pull-based approach which suits better based on our system and its particularities. As the authors of [1] explain, the pull-based approach is when the synchronization algorithm's launching is initiated from the clients and not the server. This means that each device will launch the synchronization procedure to send and receive the most recent data. Additionally, the ability to resolve conflicts comes to the hands of the users with them called to decide the correct piece of information to keep.

All of the above is a general description of the basic ideas and the requirements of the algorithm. We have studied our system's structure as well as the technique to achieve the synchronization. On the chapter following we are going to focus on the software engineering part and implement the algorithm dividing all of the work in different groups of tasks.

Chapter 3

The Synchronization Algorithm

3.1 The Synchronization Algorithm 12

3.1 The Synchronization Algorithm

This chapter presents the algorithm that is to be implemented. The algorithm in question is presented by the flow charts that follow. Each flow chart represents a phase of the algorithm and it goes with the corresponding study of it. The split was done in order to be able to focus better on the flow of the algorithm.

We have separated our algorithm into five (5) phases. The first phase is a preliminary phase in order to get prepared for the algorithm and confirm that we are allowed to proceed with the processing of data. The other four phases are separated into five (5) loops. The last two (2) of these loops are completely independent and their job is to upload and download the files that are pending. The remaining three loops (3) are connected with each other with one of them including the other two. The outer of these three loops, as shown in the Figure 4, is going through all of the database's tables and processes the two loops shown in the flow charts to follow. The two major phases that their combination gives us the synchronization algorithm are the Process Server Changes and the Process Device Changes. First to come is the Process Server Changes phase, in which all of the entries that the server has and are not yet synchronized on the device are processed. Next to come is the second phase, the Process Device Changes, in which the opposite procedures takes place. All of the new entries or modified entries on the users' devices, since the last synchronization cycle has taken place, are sent to the server. To close our synchronization cycle, we process the files that are pending to be sent to or received from the server.

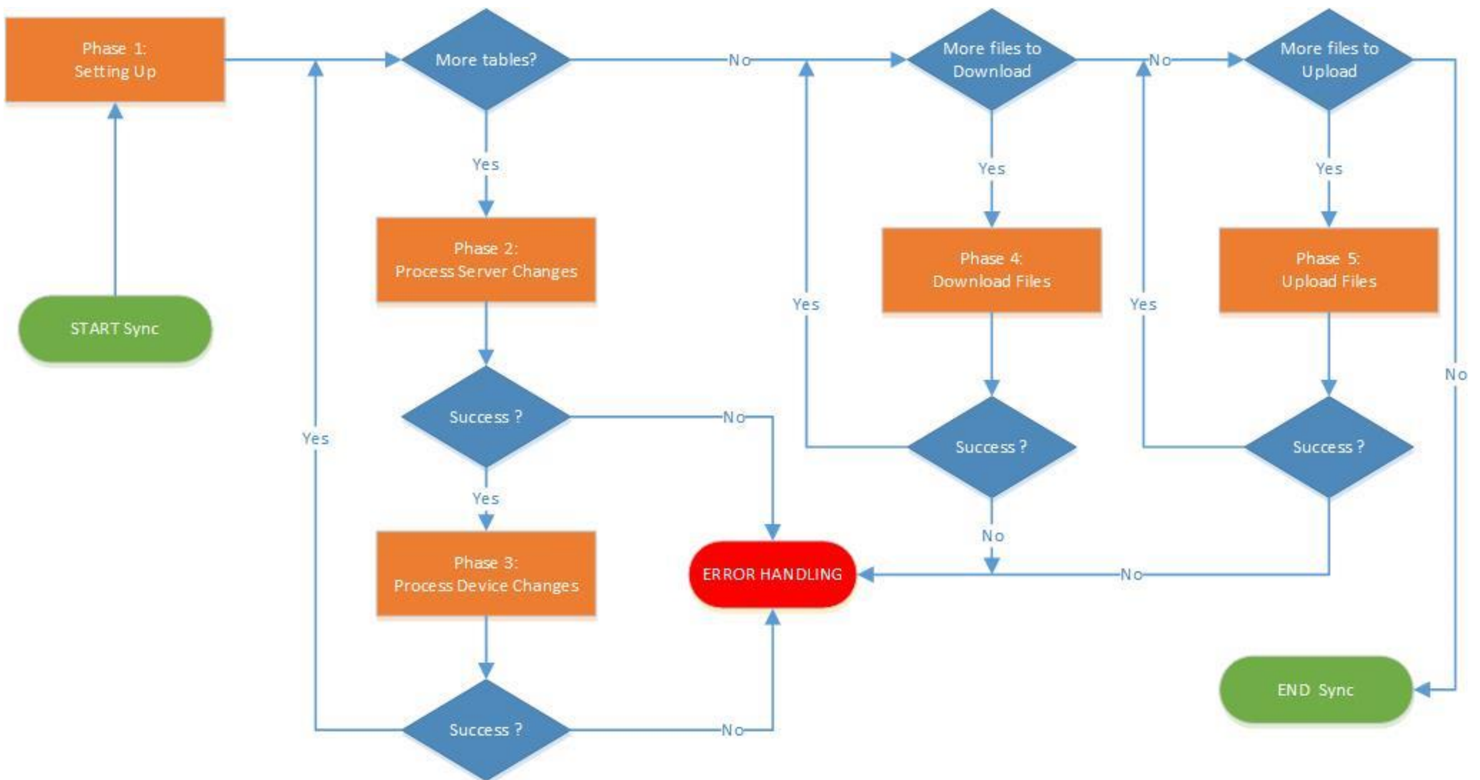


Figure 4 - Synchronization Cycle

We have studied, in earlier chapters, the usage of the Unique Sequence Number (USN) field. On the subject of this, all records maintain their USN in order to know whether they are up to date or not with the server's records. At this point, it is worth mentioning the `DbTablesSyncInfo` table. This table is created in order to maintain some essential information about each table of our database regarding the synchronization. For each of our database's tables, we maintain one record in this table with some information, with the most important being the `tableName` and the `lastProcessedUsn`. In this way, each time we want to synchronize our device with the server we know, for each table we want to process, the maximum USN we have processed so far. In consequence, we can identify changes that the server has and the device has not and process them in the Process Server Changes. On the other way around, we must identify which records are updated on the device but are not tracked on the server. To identify these records, we use the dirty bit. This dirty bit exists in all records just like the USN field and is set to 1 whenever a modification happens. This is a very elegant and easy way to find out whether there are changes on the device for which our server hasn't been informed yet. As mentioned before, this procedure is followed for each table. We have implemented the methods `getLastProcessedUsn` and `setLastProcessedUsn` in the `DbTablesSyncInfo` to set and get the maximum USN of each table. We have also

implemented the `getUnsynchronizedEntries` method that returns all of the records that are dirty for each of our tables.

Now, we shall visit the `referencedEntitiesExist` methods. These methods are responsible to check whether each entry's fields that refer to another table, e.g. foreign keys to the other table, exist. For example, assume an entry of the table Medications that has a field called `PetId` and it's a foreign key to the field id of the table Pets. If there is no entry in the Pets table with an id equal the `PetId` of the Medication's entry means that the referenced entity doesn't exist. That's what the `referencedEntitiesExist` methods do for each entry and its foreign keys. What should be done here is, that if the `referencedEntitiesExist` method failed finding the entry, then the processing must be stopped because some problem must have happened on the process of the foreign key's table and the entry wasn't updated correctly. However, there is a scenario of the foreign key's entry not sent at all. This scenario is not an actual error and it is explained in the chapter 5 during the explanation of the debugging process. In case of this failure, we stop the processing of the table and move forward. These cases are the error handling cases on our flow charts. On the following figure, we can see the scenario we have mentioned above alongside two more tables and referenced entities. The Species table, as we can see, has no references to any other tables. The Medications table references the Pets table with the `PetId` field, which with its turn references the Breeds table with the `BreedId` field and at last the Breeds table references the Species with the `SpeciesId` field.

Figure 5 - Referenced Entities

Let's now focus our attention on the Process Server Changes loop. As we can see in Figure 5, the loop starts only if the `lastProcessedUsn` is not equal to the maximum USN the server maintains. With this comparison, we can find out whether there are records on the server that are not yet synchronized on this device. We can retrieve the max USN of the server with a request to the server, we will focus on all of our requests and their structure on

Section 5.2. If the above condition is true, we move forward in our loop and request from the server all of the records that have USN greater than the `lastProcessedUsn` our device has. Meanly, all of these records are new or updated ones that our device hasn't tracked yet. For each record now we follow a certain procedure to figure out how to process this record and update our data accordingly. To process each entry, we follow the decision tree procedure below, with each indexing showing that we are moving inside the decision tree on a subtree. The two figures below, Figure 6 and 7, separate the two main paths, the entry being deleted on the server and the opposite.

Entry is deleted on the server

Case 1 - Entry doesn't exist on device (e.g. is also deleted or no existing at all)

We update the last processed USN of our table and move on to the next entry

Case 2: Entry exists on device

First, we check if the entry is clean (clean is the opposite of dirty). If clean, we can safely delete it and continue with the next entry. If dirty, we find ourselves facing our first case of conflict resolution. The algorithm cannot safely decide which copy of the data to keep and needs the help of the user to do so. After the choice is made, the selected entry is updated on both the server and the device, it is marked as clean on the device and the last processed USN of the table is updated accordingly.

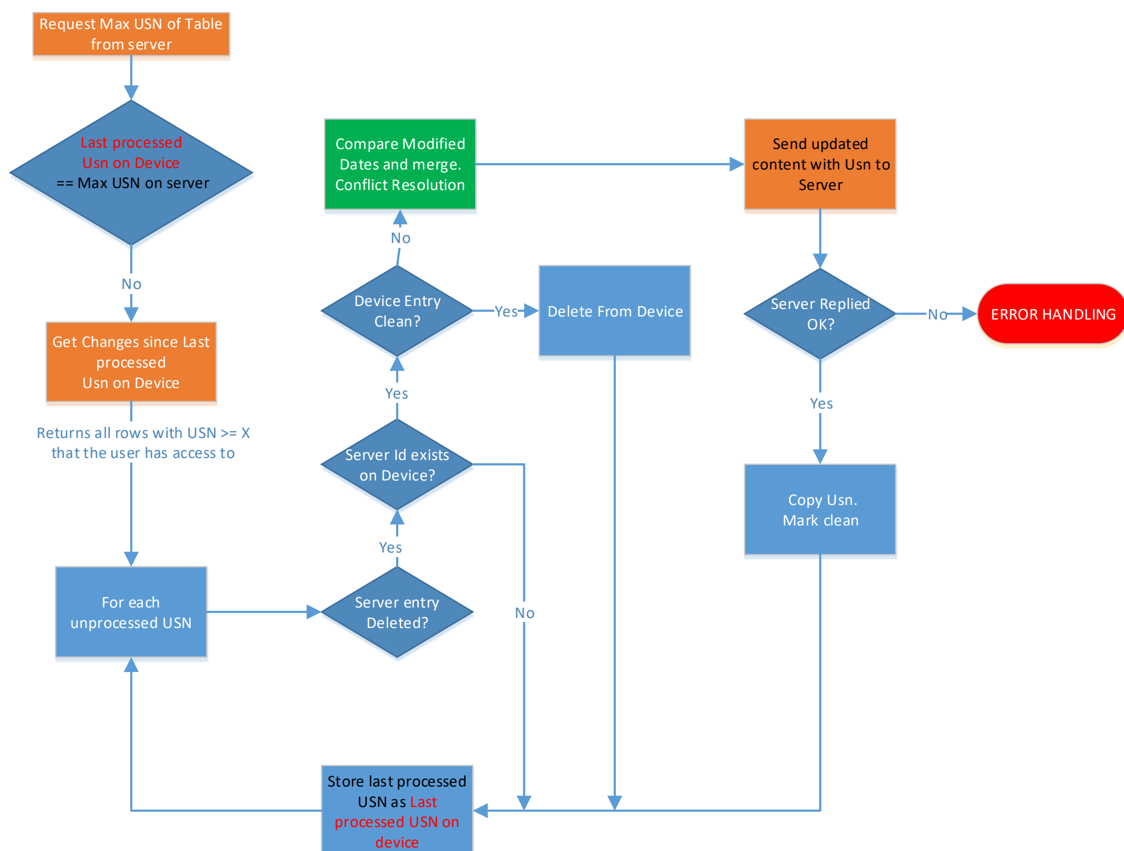


Figure 6 - Process Server Changes - Entry Deleted on Server

Entry is not deleted on the server

Case 1 - Entry doesn't exist on device (e.g. is also deleted or no existing at all)

We perform the `referencedEntitiesExist` check we mentioned before in order to see whether the entries that the foreign keys concerning the entry were insert/updated correctly. If it fails, we discard the change and stop the processing of this table. In case we continued and another entry of the table was completed with success then the maximum USN of the table would be updated. This would result in a scenario in which this entry would never be requested again from the server. The correct changes will finally come with a future iteration of the algorithm. On the other hand, if it succeeds we can write the new entry on the device and update the last processed USN of the corresponding table in order to move on to the next entry

Case 2: Entry exists on device

We again perform the `referencedEntitiesExist` check to see whether the entries concerning the foreign keys were processed ok. If it fails, we discard the change and stop the processing of this table. In case we continued and another entry of the table was completed with success then the maximum USN of the table would be updated. This would result in a scenario in which this entry would never be requested again from the server. The correct changes will finally come with a future iteration of the algorithm. If it succeeds, we now check whether the entry is dirty or clean on our device. If clean, we update the content of our device's entry with the server entry's content and after we update the last processed USN of our table we move on to the next entry. An entry is considered clean if it has not been modified since the last time it was synchronized. If dirty, we find ourselves facing our second and final case of conflict resolution. After the user chooses its preference, the selected entry is updated on both the server and the device, it is marked as clean on the device and the last processed USN of the table is updated accordingly. An entry is considered dirty if it has been modified or deleted since the last time it was synchronized.

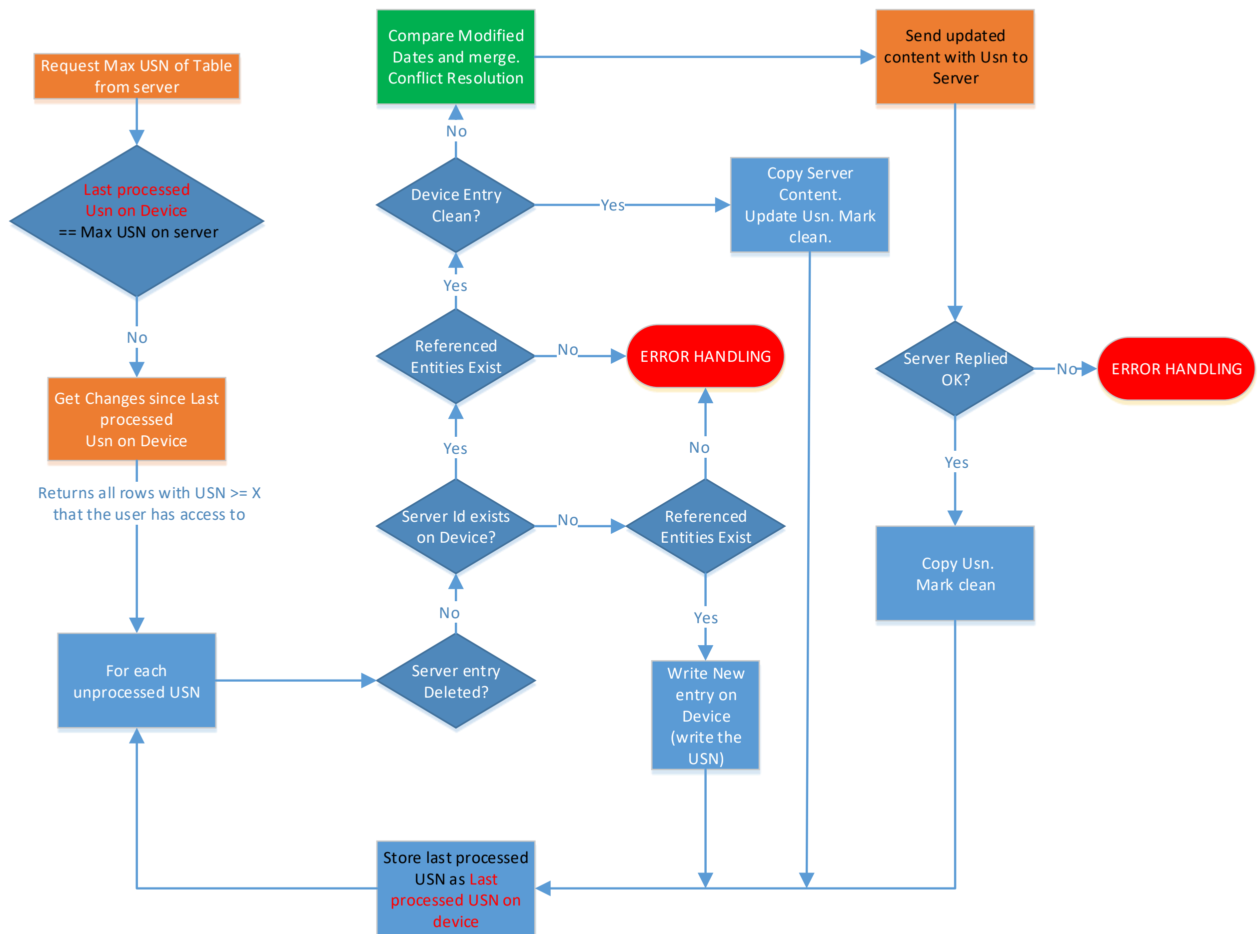


Figure 7 - Process Server Changes - Entry not Deleted on Server

The Process Device Changes loop, shown in Figure 8 and that we are going to analyze now, starts after the end of the Process Server Changes loop and it is much simpler than the one analyzed above. This loop goes over each dirty entry of the device until it processes them all. The dirty entries are the ones that were modified the devices and the server has not yet learned about these modifications. Let's have a look at the decision tree of this loop:

Entry is tracked on the server (exists on the server)

We send the updated content of the entry. We now receive the new USN from the server which we store and mark our entry as clean in order to go on with the next entry.

Entry is not tracked on the server (doesn't exist on the server)

We sent the new entry to the server. On the first communication with the server we receive only the server id given to this entry and not the USN. After receiving the server id, we now send a request to activate our entry. Then, we get a response from the server with the USN of our entry. After marking our entry as clean, we can go on with the next entry. The double communication is done to avoid having duplicate entries on the server with different USNs and server ids.

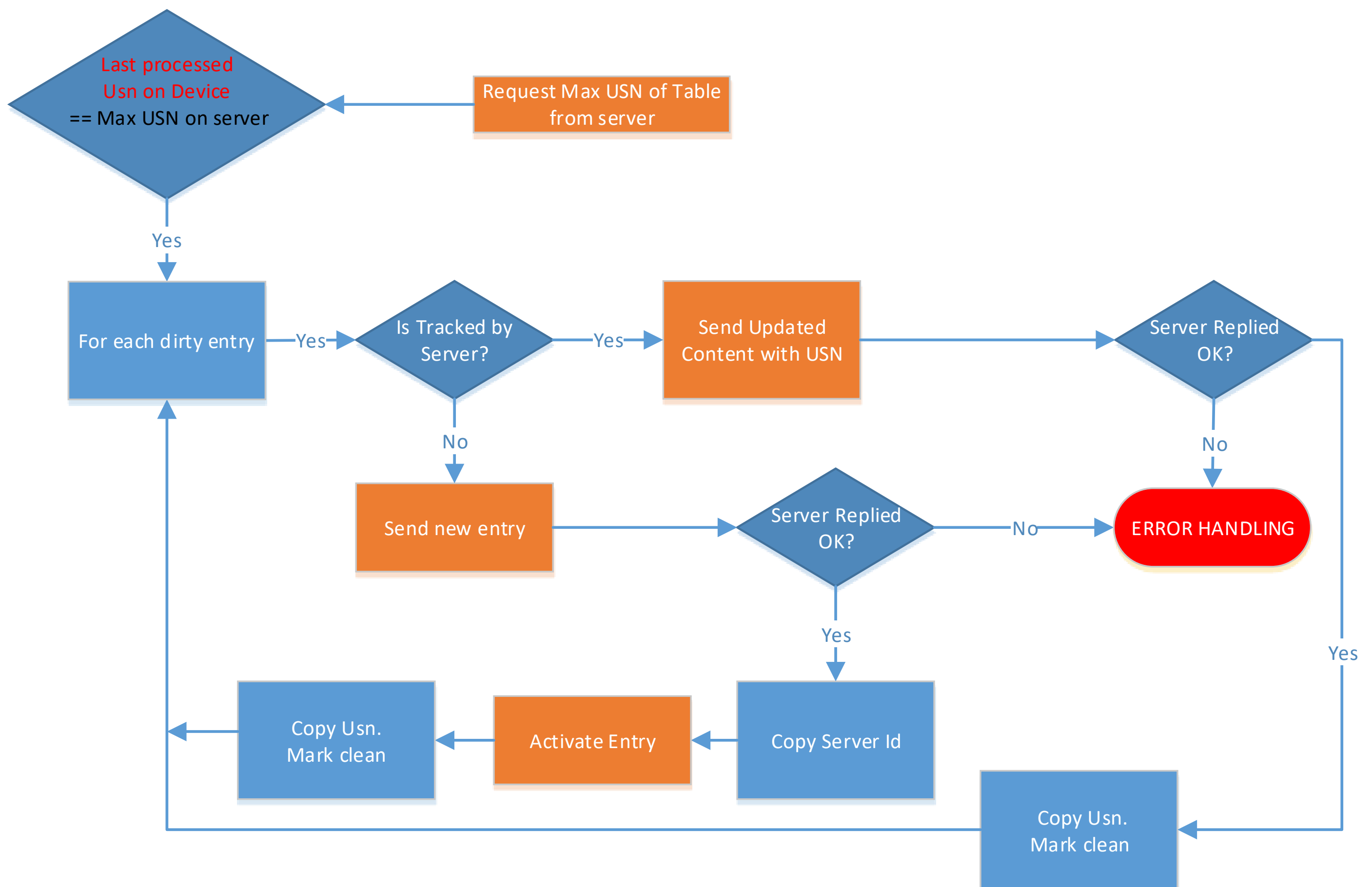


Figure 8 - Process Device Changes

Moreover, we have the Download Files and Upload Files Phases. The files of the app are stored locally on each of the devices' memory. To represent these files in the app we maintain the Media table. In the Media table, alongside some fields, we have the path field which is a String indicating the path in the device that the corresponding file is stored in order to be able to read it. To download the files from the server we must track our Media entries to whether they are pending to be downloaded or not. This tracking is done with the `pendingDownload` field. The `pendingDownload` field is set to true for each Media entry received from the server with the synchronization algorithm. All files with their corresponding media entry with `pendingDownload` equal to true are requested from the server. After their download, this field is set to false. On the other hand, to upload the files needed, we have the `pendingUpload` field in each on our media entries. All files that are added or modified have their corresponding media's entry `pendingUpload` field set to true. All files with `pendingUpload` equal to true are sent to the server. When the server receives the file, the corresponding Media entry is modified and has its `pendingUpload` field set to false. On the next synchronization cycle the Media entry will be sent on the device and the device will now have the same Media entry with the `pendingUpload` field set to

false. Next, we will study our software stack and the structures needed for the algorithm to run.

Chapter 4

Software Stack

4.1 Introduction 20
4.2 Required Structures 20
4.3 Unit Tests 24

4.1 Introduction

One of our system's particularities is the need for a "clean" algorithm from a software engineering perspective. All of this work is going to be transferred from the android environment to the iOS. Of course, each one of the two environments have its own particularities and structure. But even so, the two of them should follow the same software engineering design and stack. In this specific chapter, we will study the low-level structures that the synchronization algorithm needs. These structures are those that are going to be transferred, in the future, on the iOS as well.

4.2 Required Structures

In order to be able to execute the synchronization algorithm there are some data structures that must be implemented. The two main structures are the DAO (Data Access Object) and the DTO (Data Transfer Object). Alongside these two structures, our application already has the classes that represent the tables of our devices' database, called the Entities. The DAOs already exist in our application and they are the classes that are responsible for converting the raw data that exist in the database to the Entity object of each table's class. But even so, we are going to add a new method for each DAO that is essential for our algorithm to run. In addition to this, all DAOs extend the `SynchronizableTable` class. This class consist of a number of functions that are responsible for the updates, deletes and insertions of entries for the tables and are already implemented on our application in order to function

properly. We are going to add some new methods that will be needed for the synchronization algorithm. The `SynchronizableTable` class is implemented for all DAOs and all of its methods are generic, meaning that they can be executed by each Entity class that extends them with the same way. The DTOs are the middle classes between the server and the application. They can take an Entity object class and convert it into themselves, creating the DTO that can communicate with the server and vice versa, thus to create an Entity object class from themselves. In Figure 9, we can see how the communication is done, starting from the raw data of each device's database to the server.

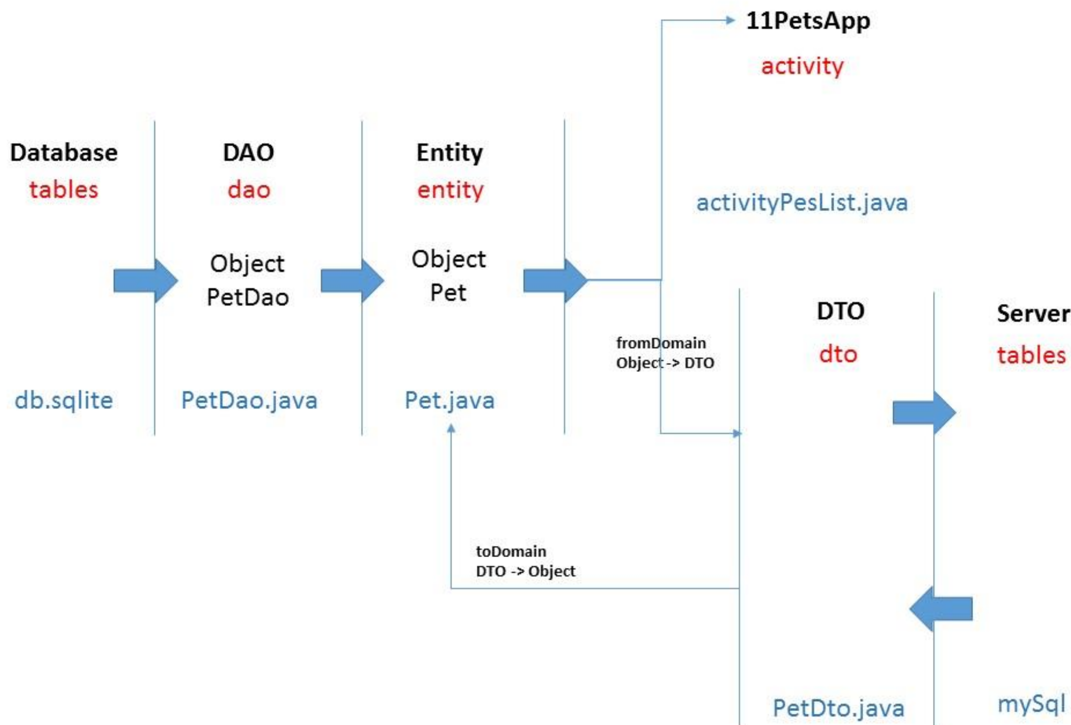


Figure 9 - Client - Server Communication

With all the above in mind let's now get into the real code and see what we have implemented for our DAOs and DTOs. All of the code we are going to present are from the Android Studio written in the programming language Java.

In each table's DAO

`deletionSideEffects` method: This method is called each time an entry of a table is deleted. It is responsible to delete all entries of the other tables that depend on the deleted entry. For example, if you delete a Surgery entry then all of the media entries (images) and note entries that are related with this surgery must also be deleted. The `deletionSideEffects` method of the Surgeries table is shown in Figure 10.

```java
//region Deletion Side Effects

protected boolean deletionSideEffects(long _id){
    String HERE = THIS_FILE + "deletionSideEffects(): ";
    if(mLOG.DEBUG){
        mLOG.verbose(HERE, "Id = " + _id);
    }

    try {
        getMediaMapDao().onDelete_host(JournalMap.HOST_TYPE.SURGERIES, _id);
        getNotesMapDao().onDelete_host(JournalMap.HOST_TYPE.SURGERIES, _id);

        return true;
    }
    catch(Exception e){
        mLOG.error(HERE, e);
    }

    return false;
}

//endregion
```

Figure 10 - Deletion Side Effects

Now you may wonder why the implementation of these methods is so important. Consider that these methods don't exist. Then all media and note entries of each surgery entry, consider the above scenario, will not be deleted and will be synchronized. This will happen for all non-deleted entries of each table's that is dependent of the deleted entry. But this is not the main problem here.

In `SynchronizableTable`

All DAOs extend this class and all of these methods are generic, namely they process and return data for each table with the same way but depending on each table's fields and structure (We will focus on the most important of them):

`prepareForSync` method: this method will be called one time for each table when the application is updated to the version that the synchronization is included. It is responsible, just like its name says, to prepare the tables for the synchronization algorithm. This includes adding UUIDS (Universally Unique Identifier) to all entries that don't have one. The UUID is a field for each entry that is unique and will help us identify same entries on the server and the device. This is important to be done because some entries that were created in earlier versions of the application may not have UUIDs on them. In addition to this, the `prepareForSync` method makes all entries that are deleted non-dirty. Just like the UUIDs, some entries on previous versions where deleted and their dirty field indicated them being dirty. This must be corrected because we don't want to send to the server and sync entries that are deleted on the devices.

`replaceCorrespondingEntity` method: the `replaceCorrespondingEntity` method is responsible to replace an entry on the device from the one retrieved from the server. This method will be called on three occasions. The first one is when we process the server changes and an entry that exists on the server is not dirty on the device. We simply replace it by using this method. The other two usages of this method is on the conflict resolution feature. It will be called if an entry that is processed is dirty on the device but the server maintains the same information about it. This may happen when a user changes an entry and sets the same data. There is no need to launch the conflict resolution, we can simply replace the entity on the device. The last one is when the user from the conflict resolution options chooses the entry from the server.

`deleteCorrespondingEntity` method: the name of this method declares exactly its use. This method is called only when the server maintains a deleted entry and its corresponding entry on the device is clean. It's important to mention that the `deletionSideEffects` method will be also called. This is done to avoid having entries that exist on the device and depend on this entry but they were not synced yet so they don't exist on the server. This means that the synchronization algorithm will not deleted on any later stage.

`correspondingEntityExists` method: we have studied our algorithm using our flow charts and we have seen that there are two points where we check whether the entry of we are currently processing from the server exists on the device. This is where this method serves our synchronization algorithm.

`isCorrespondingEntityDirty` method: as all methods we have studied, this method declares its usage only with its name. It tells us whether the entry processed is dirty or clean.

`correspondingEntityIsCleanAndNoServerChanges` method: this method is an optimization on our synchronization algorithm. It will skip an entry processed if it is clean and there are no updates on it on the server.

In each table's DTO

The DTO, as mentioned before, is the object responsible for the communication of the device with the server. Its fields may differ from the entities because the server may maintain different names for each field or may not need to receive some fields. This is what the transformation of DTO to entity and the opposite essentially means.

`fromDomain` method: this method takes an entity object of the table and transforms it and returns the DTO object of the table.

`toDomain` method: the opposite of the `toDomain` method. Taking a DTO object and returning an entity object of the table.

`referencedEntitiesExist` method: we have analyzed the concept of this method on our previous sections. This method makes reality what we have said before by checking whether the entries that this entry depends on exist on the device. As we said before, the failure of this method means that a problem has occurred on a previous table and we have to abandon this table for now and proceed to the next one.

4.3 Unit Tests

During the implementation of our DTOs we wrote more than forty (40) new classes, one for each table, with an average of three hundred (300) lines of code in each one of them. How can we validate that all of that code doesn't contain mistakes or bugs? The code is compiling successfully but we need to find a way to be sure that all of the fields of each entity are converted in the correct way into the fields of the DTO and the opposite. The Unit Tests are here to help us check the above transformation. The most important characteristic of the Unit Tests is that we can run as many of them without launching the application on any device. That means that we don't have to compile the whole app's code each time and install it on a device to check the tests. The framework of Unit Tests that was used was the Junit framework [10]. This makes the process of the testing much more quick and effective.

Each DTO is responsible to transform an Entity object to its corresponding DTO and the opposite. To accomplish this, we have implemented the methods `ToDomain`, which takes a DTO and transforms it into an Entity object, and `FromDomain` which takes a DTO and transforms it into an Entity object. The Unit Tests now have to test these two methods of each DTO class and make sure the convention is done right. Thus, for each table we have implemented a test class which consists of three methods. As expected, the two methods are called `toDomain` and `fromDomain` and it is easy to understand what they do. They are responsible to check the transformation of DTO to Entity and the opposite. This must be done by checking all the fields one by one and compare them to each other. To check whether each field of the transformation was successful we execute the `assertEquals` method of the Assert class that is contain in the Unit Tests package. This method takes as parameter two objects, in our case two fields of the DTO and Entity we want to check, and returns true if they are the same and false if not. We execute the `assertEquals` for each field of the table tested. If any of the fields fails then we can revisit our code and correct any mistakes, if all of the fields are the same then the test is successful. The third method in each of the tests is the `setUp` method. This method is executed before any test starts. Its job is to create a mock context, to be able to have access from the database and from there to read a random

entry. This entry will be transformed into the DTO using the `fromDomain` method of the DTO class. The success of these transformation will be tested in the `fromDomain` method of the test class. In case of a success we go on by using this DTO to create a new Entity object, with the use of the `toDomain` method of the DTO, and again the `toDomain` method of the test class will show us the result of the transformation.

With the above procedure being held for each one of our database's table we were able to successfully correct all of the mistakes done in our DTOs. We are now feeling confident that the server and the application can communicate and successfully exchange data with the use of the DTOs.

Chapter 5

Implementation

5.1 Introduction	26
5.2 Apache to Volley	26
5.3 Special Cases	29
5.4 Multiuser Scheme	31
5.5 Implementing the Synchronization Algorithm	33
5.6 Conflict Resolution Mechanism	38
5.7 Debugging	41

5.1 Introduction

What is to come in this chapter is the implementation of the algorithm itself. At the beginning, we will explain the process of transforming our requests to the most updated protocol that the android environment uses, the Volley. Later on, some special cases will be studied as well as the importance of the multiuser scheme. Next, the implementation of the algorithm and the conflict resolution mechanism are following. We will close this chapter with one of the most important phases of each systems lifecycle, the debugging of it.

5.2 Apache to Volley

The most common way that android applications use to communicate with APIs and servers is the Apache library [9]. Nevertheless, while the android development is evolving more and more methods of the Apache library are becoming deprecated. Deprecated, in the programming world, means that a method is not indicated to be used at any occasions because the author is not making use of it or because a better alternative exists. The most important though, is that the author is going to remove it on later updates of the library. With all the above in mind, the 11Pets company decided this is a good opportunity for us to change all of our communication from Apache to Volley Protocol [2].

Before doing this, we shall first study how Volley is implemented on android. While studying Volley we found a major difference with the Apache protocol that will make our conversion a little bit harder. The Apache is synchronous and the Volley is asynchronous. This means that there was blocking behaviour on our Apache implementation which must be implemented with a different way now. To get a better understanding of the difference assume we have the following three lines of code on Apace and Volley shown in the following figures.

```
private int foo() {
    int x = 0;
    apacheRequest(x);
    return x;
}
```

Figure 11 - Apache

```
private int foo() {
    int x = 0;
    volleyRequest(x);
    return x;
}
```

Figure 12 - Volley

Let's say that both the `apacheRequest` method as well as the `volleyRequest` method make a call to the server which returns them the value 5 and they assign x with this value. What will the value that these two methods return when they are called? Most people would say 5 but this is obviously wrong regarding the Volley implementation. The Apache implementation, due to its synchronous and blocking way of working will wait for the `apacheRequest` method to finish and then return the x which will have the value of 5. Volley, on the other hand, due to its asynchronicity won't wait the `volleyRequest` method to finish. This means that the value that the x variable has stored and will be returned will be 0. How can we solve this problem in Volley?

The implementation of execution chains will relief us from this problem and help us build our requests. The execution chains are nothing more than a list of methods. Each method in the chain does its job and when it finishes it calls the next method in the execution chain. How to know that a method has finished its job? Volley uses call-back interfaces to return the result of the Volley request. Volley requests implement two listeners, a listener that will call the appropriate method of the call-back when the request was successful and an error listener will call the method of the call-back interface on a response indicating failure. The difficulty we encountered here is the need for fully restructuring the application's communication methods. The blocking way is gone, so the volley no longer provides us the opportunity to execute more than one calls to the server in the same method. Thus, all of the code that concerns the communication with the server must be totally refactored and reordered in a new way.

Now that we know how to build our Volley requests with the execution chains let's see what kinds of features that need communication with the server we have in our app. To start

with, we have the user account features. These are the login, register and forget password. These requests only require one Volley request to the server. As a result, these features aren't so complicated to be transformed from Apache to Volley. Along with the user account features, there are some more features that communicate with our server with the use of only one Volley request. These requests include the Adopt and Blog, features of our company, the Ping feature that is used for debugging reasons and the Google features that our app supports which find nearby professionals. Though, we have three features of our app that require more than one Volley request to the server. This means that the transformation from Apache to Volley will be more complicated and the execution chains need more attention. The first two features are the backup and restore features. For now, we need to make them work with Volley until the synchronization is ready because it will replace them in order to store user's data. The following flow charts, Figures 13 and 14 show the backup and restore requests with all their methods, which is basically the execution chain. Each method does a certain job or request to the server and when it finishes it uses a call-back to call the next method in the chain.

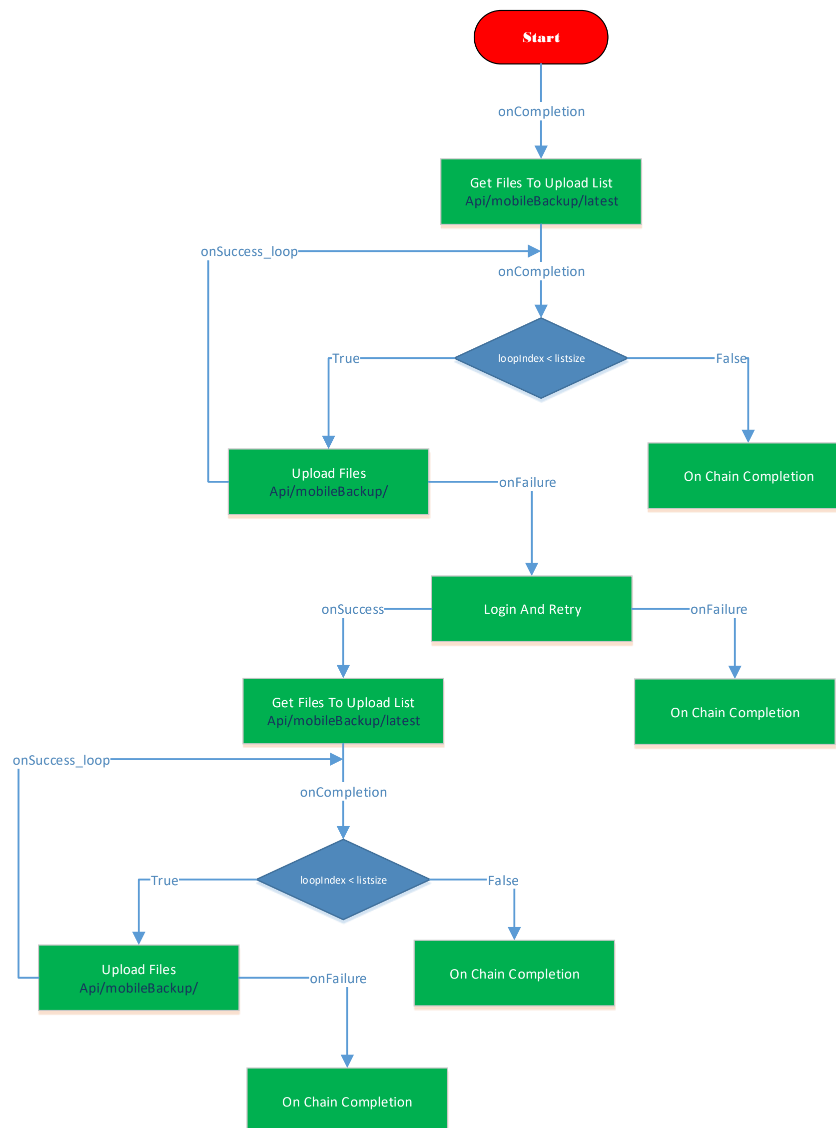


Figure 13 - Upload Files

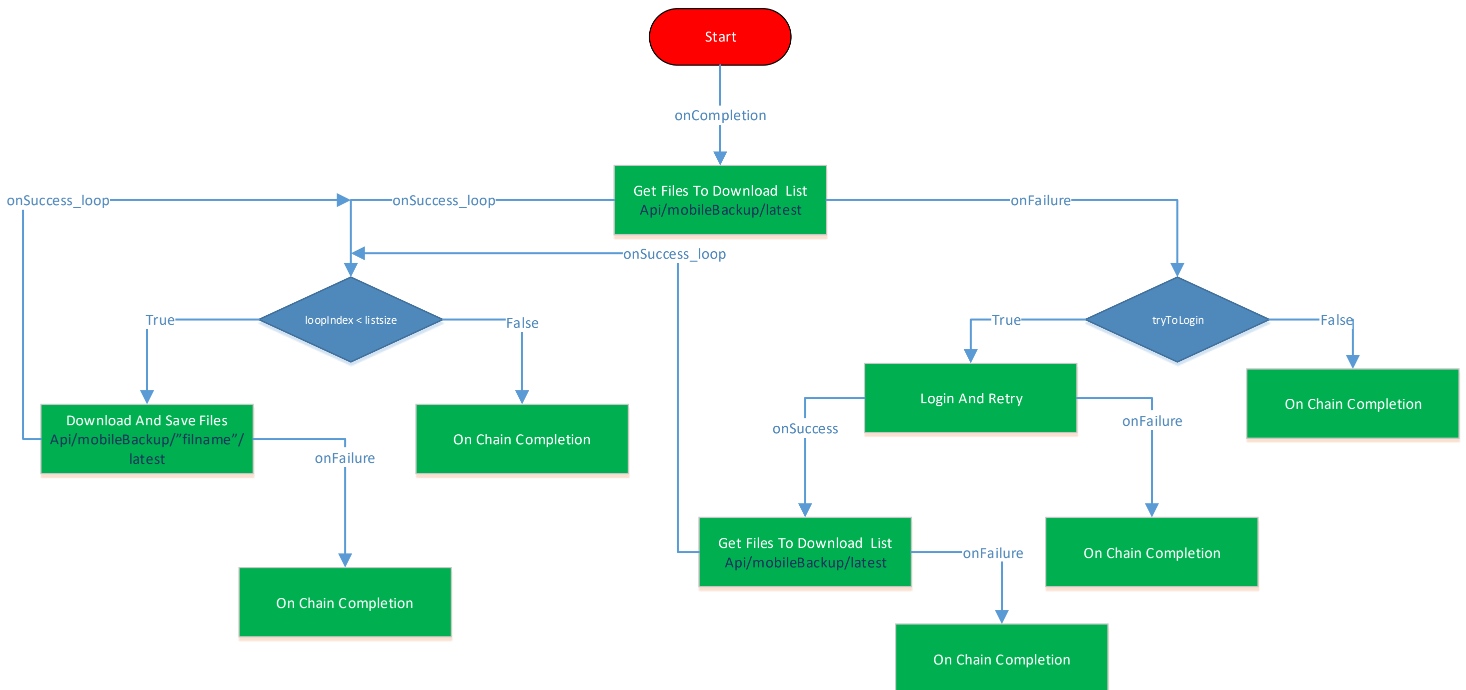


Figure 14 - Download Files

The third and final feature than needs conversion from Apache to Volley is the Upload Reports feature. This feature uploads reports, basically the logs that the app is producing, whenever an error is produced on any user's device. This is one of the most important features of the app because it gives us a feedback about bugs that happened and were not spotted while developing the app. This feature uses one Volley request for each log file it has to upload.

With all these implemented, tested and validated we released a version of the app that now uses Volley for all of its communication instead of Apache. On Section 5.5, we are going to study the implementation of the synchronization algorithm which just like all above features will be implemented with the Volley protocol.

5.3 Special Cases

To change the topic and return back to our application's structure, lets study some special cases of ours. What we have called special cases are tables of our database that are different, by some means, from the other tables and call for special handling. We are going to study the two most important special cases, the *auto-generated entries* and the *seeded tables*.

For a start, there are the auto-generated entries as we like to call them. The auto-generated entries are produced dynamically from the application itself when needed. The table that maintains these auto-generated entries is the Event Instances table. These entries

concern Food, Medication, Supplies and Daily Maintenance tables that on their term want to produce some Reminders or Notifications based on some Repetitions that the user chooses. More precisely, the application gives the user the opportunity to add single or repetitive Reminders or Notifications for each of its pets Food, Medication, Supplies or Daily Maintenance. If these Reminders or Notifications are to last forever, then of course, the application cannot produce and create all of the Event Instances entries because this would clearly be devastating. What is needed here, is a mechanism that periodically produces the next Event Instances need after the ending of the current Event Instances.

Even after the implementation of the mechanism, the synchronization algorithm came to bring up some problems regarding these auto-generated entries. Let's assume there is a Medication entry that is synchronized on two devices and produces its auto-generated Event Instances. On the one device now, the pattern of the Medication is changed. Thus, instead of let's say a Reminder on 09:00 every day we have two Reminders on 12:00 and 20:00 every day. The auto-generated Event Instance entries of the two devices would be different now. When the synchronization comes now the Event Instances are mixed up and the Medication will have scheduled Reminders both on 9:00 and on 12:00 and 20:00 each day. The solution came by adding a new table, the Epochs, which will be responsible for maintaining the same pattern for each Foods, Medication, Supplies or Daily Maintenance entry, produce the appropriate Event Instances and delete all wrong Event Instances, e.g. those that don't match up with the specific Epoch.

Secondly, there are the seeded tables. Seeded tables are the tables of our database that are completely or partially the same for each user. For example, the Species table is a completely seeded table. There are four Species entries, the dog, the cat, the rabbit and the other. They are the same for all databases and no one can add, delete or modify any entry. Another example, this time of a partially seeded table, is the Maintenance Type table. This table, describes the daily maintenance of the pets like, bath, ear and teeth cleaning and more. There are numerous seeded categories but the users are free to add any new Maintenance Type they want. The server, on its hand, keeps all of these seeded entries one time, meaning that all of the users share these entries with the server and each other. To achieve the above, the seeding of these entries inserts them in the user's databases with a static UUID which results in being the same as the server entries.

As already mentioned, some of our seeded tables can be characterized as partially seeded because the users can add new entries on them and modify or delete those entries. The problem comes with the Maintenance Type table, a table on which the application gives the user to modify the seeded entries. The modifications correspond to whether the Maintenance Type would be visible to the user and the position of the Maintenance Type on the

corresponding list. Following up our earlier statements about synchronizing the seeded entries, any change of a seeded entry on one device would be propagated to the server and with its turn to the other devices when they synced! This scenario is more than unwanted for our system. Just like the problem with the auto-generated fields we decided to add a new table for this problem as well. Therefore, we created the Maintenance Type User Specific Fields table that carries out this task of keeping the modifications made on the Maintenance Type seeded entries. Whenever a modification is made on the application regarding the seeded Maintenance Type, the information coming with this modification is stored in the new table and respectively it is fetched from it whenever needed. At last, this results in a safe way of synchronizing the Maintenance Type and the Maintenance Type User Specific Fields tables without affecting all users and databases. In Figure 15, we can see the Maintenance Type User Specific Fields table containing the fields that can be changed from the user regarding the Maintenance Type table and referencing it with the `MaintenanceTypeId` field.

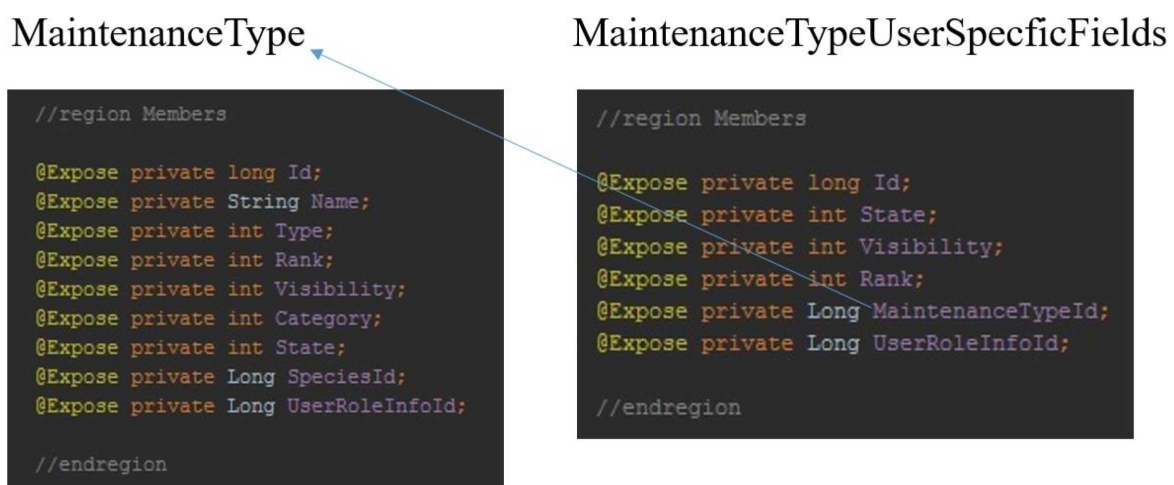


Figure 15 - Special Case - Seeded Maintenance Types

5.4 Multiuser Scheme

With the arrival of the synchronization algorithm, our application should be refactored, both on its database schema as well as its inner implementation in order to support multiuser usage. A Multiuser scheme is a system on where more than one users can login on the same device without their data being affected on any reasons. A Multiuser system gives the illusion to its users that each one of them is executing alone [7]. Before the synchronization, all entries stored in the application's database were considered to belong to one user only. Well

of course, this is contradicting with idea of synchronizing and be able to transfer each user's data onto more than one devices.

To achieve the Multiuser Scheme two major tasks were needed. Firstly, the database schema should provide a way of categorizing the entries based on which their owner is. For this reason, we added the `dbOwner` field in every entry of every table of our database. This field is a foreign key to the User Role Info table's id field, in order to know which of the users stored in the database its owner is. Also, the User Role Info table needed modification because up so far only one entry existed and was overwritten each time someone logged in the device. From now on, each time a new user logs in from a device, a new User Role Info entry is created and all of the entries added on the application, while this user is logged in, are added with the `dbOwner` field being the id of this user. Finally, we went into the application's code and modified all of the reads and inserts to include the `dbOwner` field. Meaning, that all of the entries insert include the corresponding `dbOwner` and only the entries with the corresponding `dbOwner` field are read and retrieved from the database. The Multiuser scheme, gives us the opportunity to the user to login from any device and retrieve its data. In addition, when logging out his data stay safely hidden in the application without others users of the device being able to modify or read them. In Figure 16, you can see a graphic representation of a database maintaining the data of three (3) users.

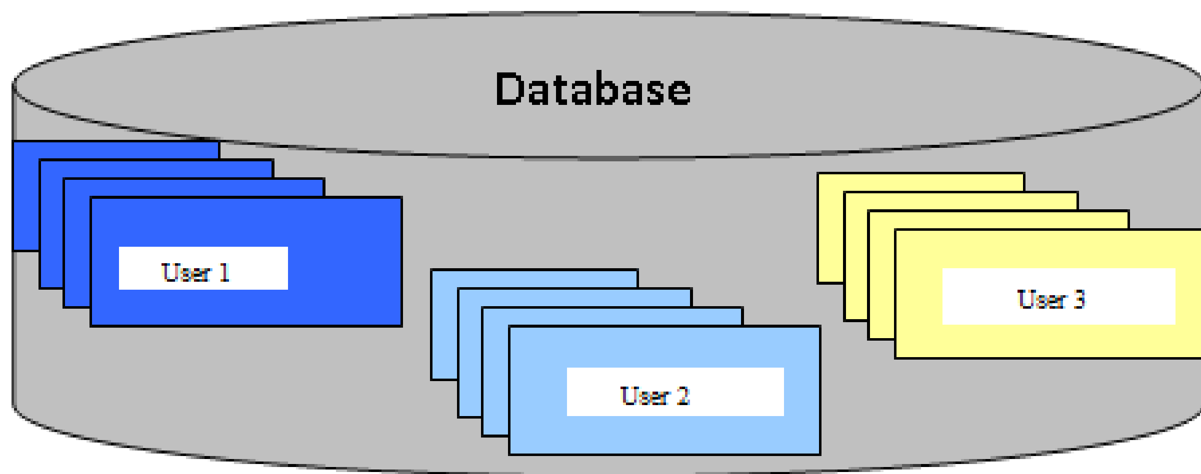


Figure 16 - Multiuser Scheme

5.5 Implementing the Synchronization Algorithm

To continue, we have so far studied our algorithm in depth and we have implemented all the necessary structures, and tested them, that the synchronization algorithm requires. Let's then get our hands dirty and start writing down the code of the algorithm based on what we have seen so far. To make our algorithm easier to study we have split it into four phases. Each phase represents a group of requests that carry out a feature of the algorithm. It is not hard to understand that the two phases are the Process Server Changes and the Process Device Changes that we have studied on Section 3.1. Those are the Phase B and Phase C of the algorithm, respectively. Phase A is the Setup part of the algorithm, namely all of the requests done to determine whether the synchronization is feasible. The last phase, Phase D, is where all files will be synced. These files are the images that each user has stored in his own device. For each phase, we will present a flow chart, with all the methods of the execution chain, and explain for what each method is responsible. The colouring of the flow chart isn't random. The methods with green colour are the methods that execute a Volley request, the orange colour represents methods that don't communicate with the server, the blue rhombus represent if statements and the circles, blue or purple, tell us where to go next. The method `onChainCompletion` is the method where we evaluate our outcome. It can be called either when an error occurs and the sync will be stopped, the outcome will be the appropriate error or when the sync finishes with success and the outcome will be the success of our synchronization. We are now going to visit each phase and study it in depth.

Phase A – Setup

The Setup phase, shown in Figure 17, is here to execute all necessary work that is needed in order to be able to execute the synchronization algorithm. Firstly, the user must be logged in in order to execute any requests. Following the login request we have the sync gate feature. This is a way to control whether the device is able to pass the synchronization gate or whether it will be stopped. The sync gateway consists of two version numbers that are checked. The first one is the database version of the device itself and the second is the server version that the app is implementing. For example, if the app implements version 1 of the server and the server is now on version 2, then it won't allow the device to pass the gate and synchronize. Also, if the device's database is version 3 and the server requests a database of version 4 and above then again the gate won't be passed and the synchronization will not be allowed. Next to follow is the registration of the device. This request registers the device to the username of the user that tries to synchronize. The following two methods, the

`setDbOwnership` and `checkDbOwnership` have the responsibility to either confirm that the device is owned by the account trying to sync or claim it if it is the first time this device is being synchronized. If case of an account trying to synchronize a database that belongs to someone else this will result in an error. The success of the above will transfer our algorithm to the Phase B.

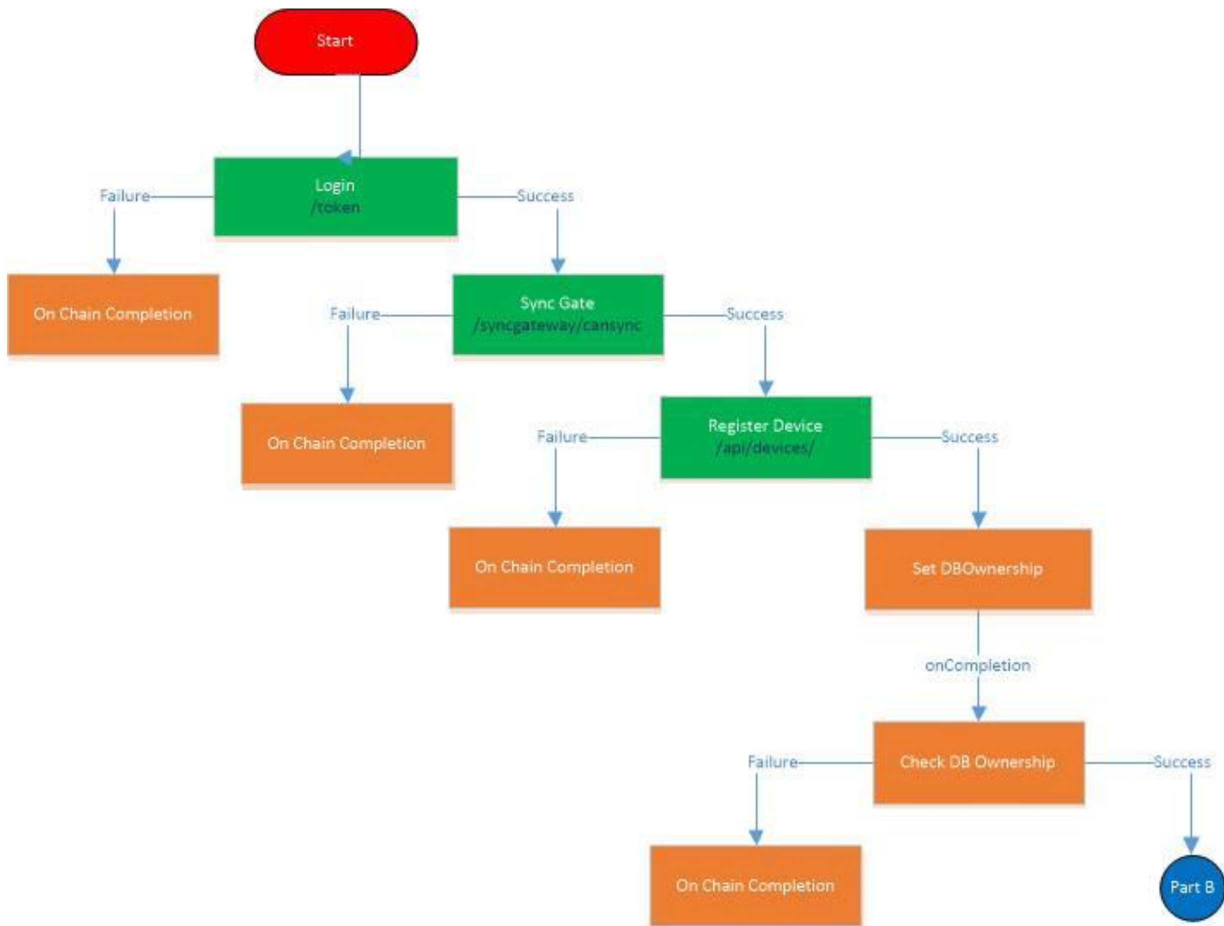


Figure 17 - Phase A - Setup

Phase B – Process Server Changes

The second phase of the synchronization algorithm is the Process Server Changes phase. We are familiar with this phase because we have seen it again when we were studying our algorithm on Section 3.1. The first thing this phase does is to initialize the table that is going to sync. The initialization of the table that is going to sync is basically to set the correct DAO and DTO classes alongside the correct names for the current table. After this is done, we move on and request the server USN of the table from the server. We then read the device USN of this table from our device. This is where the comparison of the server USN and device USN is done. If server USN is not greater than the device USN we check whether we are done with this table, if yes then we continue with the Phase C, if not that means he have

left entries unprocessed and we will return back to get the server USN. On the other hand, if the server USN is greater than the device USN we request the server changes that concern our account and that have USN in the range of device USN and server USN. For each of these changes we follow a certain procedure to process it. This is done in the process server changes method. This procedure is the one we discussed in Section 3.1. In Figure 18, you can see the flow of the Process Server Changes phase.

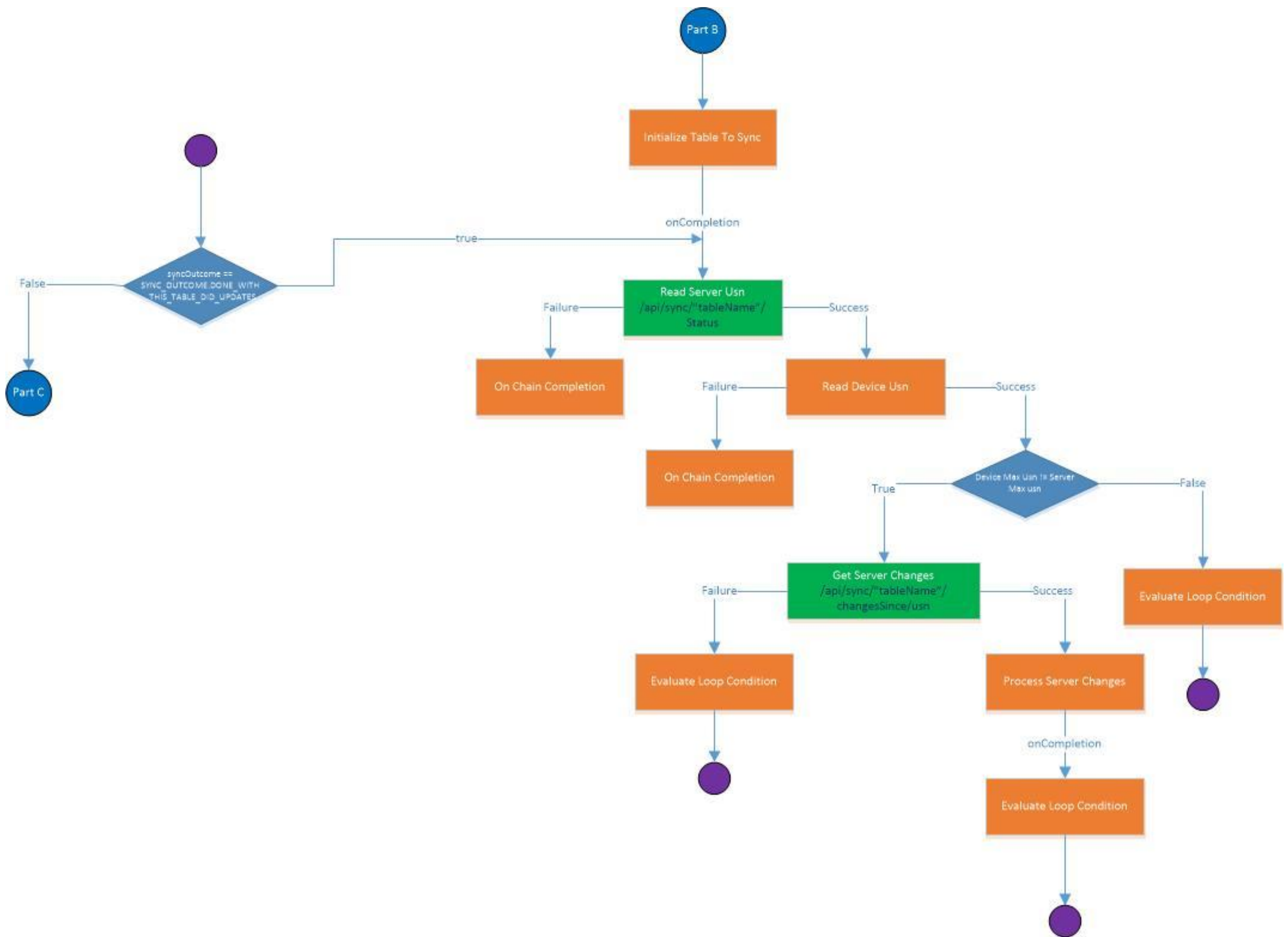


Figure 18 - Phase B - Process Server Changes

Phase C – Process Device Changes

With the server sending all its changes to the device we shall go the opposite way now, send the device changes back to the server. As Figure 19 shows, firstly, we evaluate the result of the loop in Phase B. In case of an error we stop the sync, if all are ok we continue and with the `initUnsynchronizedList` method we fetch all of our changes that must be send to the server, these are all the dirty entries of our database. For each of our unsynchronized entries we have two possible paths to follow depending if the entry is tracked by the server. For existing entries, we send a Volley request with the updated entry. For the new entries, we first send the new entry with a starting request and then another request follows to activate the entry.

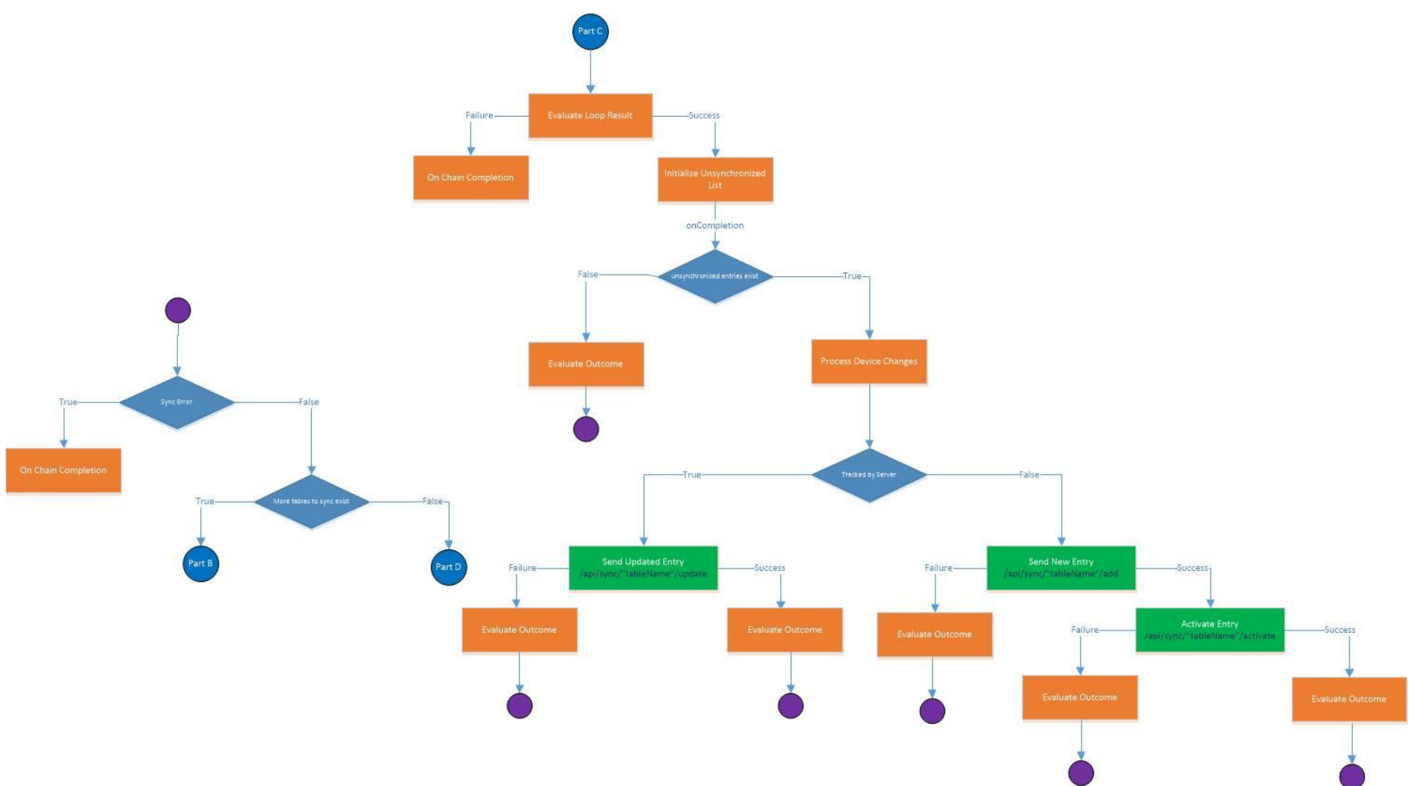


Figure 19 - Phase C - Process Device Changes

Phase D – Process Files

The 11Pets app gives users the opportunity to add images of their pets. Each image is described in our database as a Media entry. These images, of course, cannot be stored in the database but they are stored in the device's storage. We have to make sure that we have a way to synchronize these files as well because they are very important for the users. To achieve that we added two new fields in our Media table, the `pendingDownload` and the `pendingUpload` Boolean fields. The `pendingDownload` field is only kept in our local database. On contrast the `pendingUpload` field is held at the server as well the server is

the one that modifies it. The `pendingDownload` field is set to false for each file the user already maintains at the local storage. Each time a new Media entry the server sends us we set its `pendingDownload` field to true. We also set a Media's entry's `pendingDownload` field to true if we get an updated Media where the update concerns the file. On our algorithm now, when we get the files to download from our `getFilesToDownload` method we get all Media entries with `pendingDownload` true. After we request and get these files we set the corresponding Media's entry's `pendingDownload` to false. On the other hand, we have the `pendingUpload` field which is set to true for each Media entry that has not yet synced with the server. The `pendingUpload` field is set to true whenever we change an existing Media's entry's file or we add a new Media entry and file. We are going to send to the server all the files that have `pendingUpload` true in their corresponding Media entry. This is done in the `getFilesToUploadList`. Whenever a file is successfully sent, the server is responsible to inform us so we can set the `pendingUpload` field to false. The server needs this field in order to know from which Media entries there are files to expect. In Figure 20, we can see the flow chart of the download and upload files from and to the server.

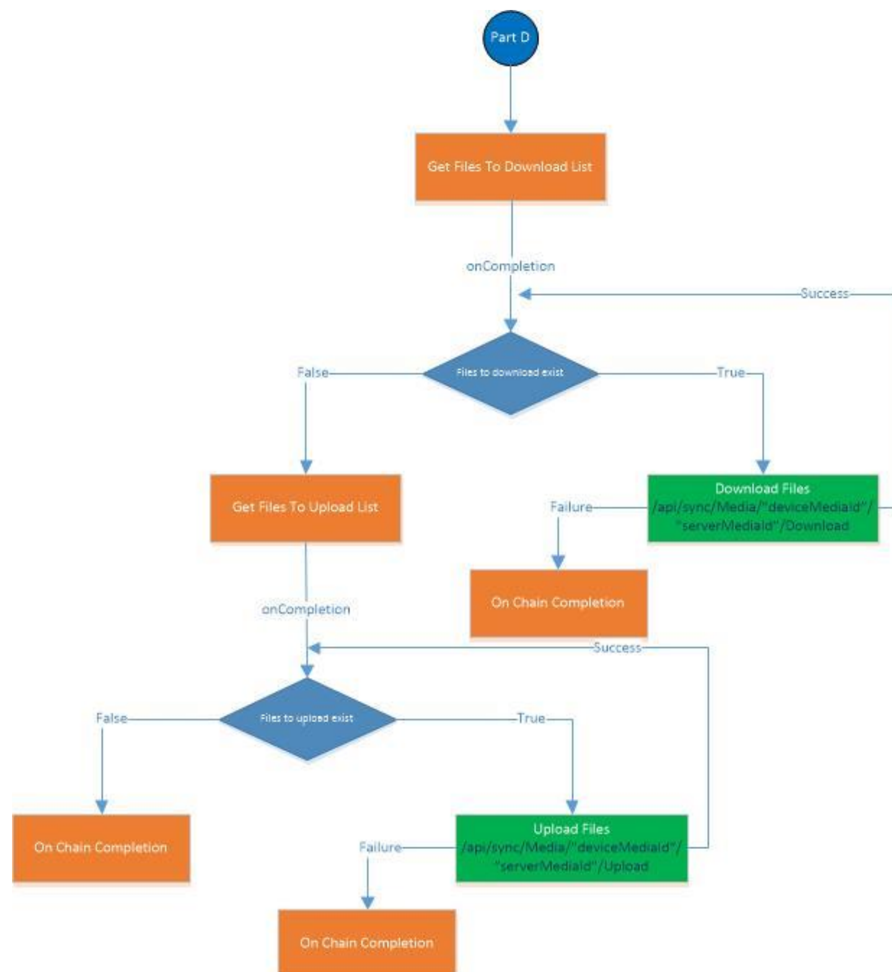


Figure 20 - Phase D - Process Files

5.6 Conflict Resolution Mechanism

Having data conflicts when synchronizing data between two or more devices is inevitable. What plays important role about each company is how they are handled. It is not unusual to share data with other people and never come across conflicts. But, this doesn't mean no conflicts happen, instead it means that the application you are using resolves conflicts on its own without asking you, more possibly keeping the most recent piece of information for each case. As stated earlier, we have decided to provide three options for conflict resolution:

- Keep the local entry of the device.
- Keep the entry that the server has.
- Keep the last updated entry.

In this section we are going to study the occasions that conflict might happen on our algorithm, some optimizations we added and the classes and interface of our mechanism.

Conflicts can occur on our app when two or more devices, sharing the same account, modify the same version of an entry and try to synchronize. More specifically, assume two devices with the same account that are synchronized and maintain the same piece of information. Consider an entry of a pet that has the name X. The first device changes the entry and sets the pet's name to X1 and the second devices sets it to X2. The server maintains the entry of the pet with the name X so far. The first device now synchronizes and updates the entry on the server with the new name, X1. The synchronization results in the update of the USN of the entry as well. When the second device starts the synchronization algorithm notices that there are entries on the server with greater USN than the device's USN. The server sends this entry and that's the point where we find out that the corresponding entry on the second device is dirty! The above scenario can happen also in a case of deletion of the entry from one of the devices. For example, the entry might have been modified on the first device and deleted on the second or the opposite. Again, when the server sends the entries with USN greater than the local one we will locate the conflict and try to resolve it. Based on our algorithm's design, the only part that conflicts are spotted is Phase B – Process Server Changes. The algorithm's logic and design guarantees us that this is the only place that we will find conflicts. So, there is no other part of our code from where problems may occur due to conflicts.

Before releasing the algorithm, we tried to figure out any optimizations that we could do in order to launch the conflict resolution mechanism only when necessary. We came up with a neat idea to avoid a group of unnecessary conflicts, the conflicts occurred but concern the

same piece of data. Based on our previous example, consider than both of the devices change the pet's name to X1. For the algorithm, it doesn't matter if the entry has the same piece of data, it is still dirty on the device which will trigger the conflict resolution mechanism. This might be a valid conflict but the users don't know how the algorithm is implemented and they will be asked to resolve a conflict among two entries that are exactly the same! The optimization comes to help us identify these cases and resolve them automatically without the user knowing anything. We implemented the `sameObjects` method for this reason. This method takes two objects as parameters, the local entity and the remote one. For each one of them it calls the `getFieldsListForConflictResolution` method (analyzed in next paragraph) and now we have the two lists with the fields that must be checked. This fields are `ConflictObject` objects that we are going to study on the next paragraph. The method checks one by one the fields of the two objects to find whether they are the same or not. If all the fields are found being the same then we come to the conclusion that these two entities are the same and there is no need to trigger the conflict resolution mechanism. If at least one field of the entities differs then conflict resolution is really needed. There are two more optimizations that we discovered while debugging our algorithm. These optimizations are going to be presented in the following subsection, the debugging of the algorithm.

We have seen all the theoretical part of the conflict resolution but we have not yet written any code that will handle them. Firstly, we need to create a class that will be able to describe the fields of the database in order to be checked for conflicts. The `ConflictObject` class was created for this job. This class is consisted out of 6 fields:

- `Label` – String: description of the field to help the user understand better that each field represents
- `Value` – String: the value of the field in a string datatype. For example, if we have a field that is integer and corresponds to an enumeration, here we will show the value of the enumeration in the index of the integer that this field maintains
- `ValueString` – String: the value of the field – used only for String type fields
- `ValueLong` – Long: the value of the field – used only for long/integer type fields
- `ValueFloat` – Float: the value of the field – used only for Float type fields
- `Type` – Enumeration: values: String, Long, Float, Date. Determines which of the above values will be used, the other two remain null and won't be used by the object. The Date type was added after the debugging of the algorithm and it is one of the two optimizations we are going to study later. The Date type uses the `valueLong` field but it is treated differently than the Long type.

The ConflictObject has four constructor methods, one for each of the above types. The most important method of this class is the compareToByValue. This method takes a ConflictObject object as a parameter and compares it with the ConflictObject object on which it is called on. The comparison is made based on the type of the object on which the method is called. If the object that was send as a parameter is a different type of object, then the comparison stops and an error is produced. If the two objects are of the same type, then their values are compared and the method returns true if they are the same and false if not. As mentioned before, when a conflict is spotted, the local and remote object create a list of ConflictObject objects with each object representing one field of the object. This is done in the sameObjects method and by calling the compareToByValue method on each of the ConflictObjects of the list we can find whether a different exists an on what field of the entry. Another class that is used in the conflict resolution mechanism is the SynchronizationConflict. This class is created after the conflict is spotted and verified and it contains the two objects, the local and the remote, and the name of the table that the conflict's entity belongs to.

Having mentioned the getFieldsForConflictResolution, we shall refer to where it is contained. We will also have the getContextForConlflictResolution method that is needed for the synchronization algorithm and goes with the getFieldsForConflictResolution method. These two conflict resolution methods are created in each of the Entity classes of each table of our database. The first one creates a list of ConflictObjects and returns it to the caller. It is worth mentioning that we don't create a ConflictObject for all of the fields that the Entity maintains. There are some fields of each table that doesn't make sense or are not needed to be included and checked in the conflict resolution mechanism. For example, for a Surgery entry there is no need to create a ConflictObject for the petId field because this field cannot be changed. In case of different petIds in two objects that are said to be the same, we can say that we have a more general problem in our application and not in the synchronization algorithm specifically. The second method, the getContextForConlflictResolution returns a pair of Strings that are used in the activityConflicts, the activity class that is shown to the user and takes the users input, as the title and subtitle of the conflict. For example, for a conflict of a Surgeries entry, the title returned is the name of the surgery and the subtitle is the name of the pet that the surgery refers to.

How will we present the conflict to the user and help him decide how to resolve it? The activityConflicts is the activity class that is responsible to present the conflict of

the algorithm to the user and guide the user to the correct decision. As you can see in Figure 21, the toolbar of the activity consists of the title and the subtitle. Some tables may have only a title. The main screen has two columns, the local entity column and the remote entity column. Each row of the columns is a `ConflictObject` object and all of the rows constitute the list we described earlier. At the bottom of the screen, there are the three choices we have given our users on how to resolve a conflict: keep the local entity, keep the remote entity or choose the automatic resolution that will keep the most recent updated entry.

Figure 21 - Conflict Resolution Interface

5.7 Debugging

Probably the most important part in the implementation of any system is the debugging of it. When debugging, a developer has the opportunity to check all aspects of the system created and all the corner cases that a user may produce. While debugging, someone has the chance to spot the most important and bad bugs on the system. Of course, not all bugs and errors in any system can be found while developing or debugging. That’s where error logs and reports come in hand, to inform the company for corner case errors that were not spotted without the real users and data.

The debugging of our algorithm is divided to three phases. The first one was done with fake data and without the algorithm being live in the users' devices. This phase's purpose is to spot the large majority of the mistakes. Mistakes that create errors that cannot be avoided during the synchronizing of a simple database and are very common. During this phase the first synchronization cycle was launched. Generally, the first synchronization cycles can be characterized as successful. The study in depth of the algorithm and the careful implementation of its needed structures and the code of the algorithm contributed in this result. In addition to this, we were able to find and correct mistakes that were made on the conversion of the entities. For this, we have already done a lot of Unit Tests but those tests were done on static objects and we have missed out some cases. This phase was also very helpful to us on developing the Conflict Resolutions interface and methods. At the beginning, all the fields of each Entity were shown during the conflicts and the conflicts testing showed us that this will not be good for the actual users. So, we left the fields that really interest the user and that make sense to the knowledge a user has for the data. In addition to this,

The second phase was the live data one. The algorithm was released and was running on the background of the users' devices and synced their data without users manually triggering it. This triggering of the synchronization algorithm would happen ten seconds after a user proceeded to any change that resulted in a write operation on the local database. Coupled with a mechanism that prevented more than one synchronization algorithm to be launched we were good to go. Of course, we asked the users for their permission before proceeding with this feature! Also, the synchronization would only run if the user has done a backup of the database with the last version, to prevent any mistakes of our algorithm to result into data loss and also run only on WIFI connections so we won't use the data of the user for our purposes. Additionally, we disabled the conflict resolution cases and we added an error logging mechanism when conflicts were found by the algorithm. With the one way synchronizing we wouldn't expect any conflicts to actually occur but thankfully we added this logging and as we are going to see later, conflicts did occur! During this phase, we were able to discover a very important bug that we couldn't have spotted while the local debugging. We received a numerous of errors reporting that devices tried to synchronized Event Instances entries and Repetition entries while having their `RepetitionId` and `PetId` null respectively. This was spotted by the `referencedEntitiesExist` method which found the referenced Repetitions and Pets entities deleted. This error was a bug that was created during a precedent release that was responsible to delete all of these entries. Obviously, this release was defective and the debugging on the real data gave us the opportunity to correct this wrong.

Now we shall refer to the two optimizations of the conflict resolution mechanism that were discovered during our debugging. Firstly, we received many conflicts that occurred on the date field of entries. Why did this happen? On both our app and server, the dates are saved as long types and they are the timestamp of the date that they represent. The big difference is that the server doesn't keep milliseconds, this results in the last three digits of a date converted to a timestamp from the server (the `ToDomain` method is responsible for this) to be equal to 0. Consider now the scenario where a Surgery entry is synchronized with the server and the date is stored with the three last digits to be different than the apps ones. When this entry is modified and the synchronization goes through the Process Server Changes phase, this entry will be found as dirty and its data will not only differ on the field changed but also on the date. To solve this one, we added Date as the fourth type of our `ConflictObject` object. While converting our date field to a `ConflictObject` we set this last three digits as zero so they will not differ from the server's. Although we added this optimization we noticed conflicts to be reported, this time not only for the date field but on other fields as well. How could this happen? While taking a better look at our algorithm and the reports we could identify what was causing these conflicts to emerge. On a first synchronization cycle, a new entry will be sent to the server during the Process Device Changes phase. The update of the device's USN field though, for each table, is done in the Process Server Changes phase. So, when a second synchronization cycle was launched the server sent that new entry we referred to before and the expected result was to be the same on the device and the update of the device USN would be successful. What if the entry was modified before a second cycle of the synchronization was launched? That was the case we didn't consider because the design of the algorithm basically needs two cycles to be complete. To solve this problem, we decided to use the device id property. Each time an entry was sent or updated from the device to the server during the synchronization, the server would keep a field indicating from which device this transaction was done. Then, when the server sent the entries back to the device, the device id that was stored before was sent alongside the entry. If the device id was the same as the id of the device receiving the changes then we know this is no conflict scenario and we should not do anything. The entry of the device, being dirty, will be sent to the server on the Process Device Changes phase and all will be good.

At the same time, a large group of non-existing referenced entities emerged. After taking a better look at the errors we could identify that all were produced by the same user and during a short amount of time. Following this observation and the error logs we were able to understand what was going wrong. Let's assume we are a user and we want to add two new pets in our application. Whenever a new pet is added, the application creates new

entries for more than the Pets table. The entries for its maintenance are created, the repetitions of any events that are automatically added, the Pet Extra Info which is used in case of the pet being a father or a mother to another pet and more. Not to mention that all these tables are processed after the pet table is processed. Considering this, we add our first pet and after ten seconds the synchronization algorithm starts. Just when the synchronization completes the Process Device Changes for the Pets table we add our second pet. It's not hard to understand that the second pet was not sent to the server because it was added after the pets table was processed. As a result, when the algorithm moves on a table we mentioned above, the entries of the second pet will be there and we will try to sync them. However, the referenced entities of the pet won't exist and we have ourselves the bug. This bug can be characterized as a fake error because on a second synchronization algorithm all will be done normally due to the fact that the pet entry will be synced firstly. The problem now, lies on the fact that this can either be a fake error, just like the one described above, or a real case of a referenced entity missing. Consequently, there was a need for a mechanism to clarify the fake and the real errors. Based on the idea of the fake errors to be those that occur once, we came up with a fast and easy solution. Each table in our database maintained a field called ffu01 and was null for all entries because it was not used for any reasons. Therefore, we used that field to mark any entry that created the referenced entities problem. Whenever any marked entry confronted the referenced entities for a second time an error report would be produced indicating a real error. Each time a marked entry was able to find its referenced entities and sync properly we would unmark the entity back by making the ffu01 field null again. For our own good, this never happened and the mechanism worked.

Eventually, we move forward to our third and last phase of debugging. This phase is more of a verification of the second phase's success. This is where we should be able to get a better view of whether our synchronization algorithm was successful. The procedure is simple. Retrieve a numerous of user databases from their backups and push them into a device of ours. Then login the same user's account from another device and try to synchronize all their data. If our algorithm is successful, the two devices should have the exact same piece of information. All of the above of course will be done at a replicate server of the original and will not affect at any point the real data and databases of the users. After testing a numerous of databases, we came across a strange phenomenon that just needed a moment of thinking to be solved. All of the data were transferred correctly, but the schedule was empty. No food tasks, no medications tasks, no hygiene task nothing! Just a second of thought was needed to understand that this was absolutely natural. All of these tasks on the schedule of each device are Reminder entries which are produced based on the Medications, Food and other entries that result into Event Instances. The Reminders though aren't

synchronized because they are time specific and each device is responsible for its own Reminders. For this reason, a call to the appropriate method of the Reminders Service created all the schedule and all were back to normal. To conclude, our third phase was a success without any errors being produced. As a result, we are ready to release the synchronization live and inform the users so they can start taking advantage of it and its advantages over the simple backup / restore operations.

In summary, our synchronization algorithm is ready to go live! We have carefully implemented it, debugged it very carefully and treated all the necessary corner cases accordingly to make our algorithm more efficient and secure. Following up, there will be an explanation, like a brief manual, on how the synchronization feature actually works on the devices.

Chapter 6

System Usage

6.1 Introduction 46

6.2 System Usage 46

6.1 Introduction

The 11Pets application offers a plethora of features to its users for taking care of their pets. In this chapter, we are going to create a brief manual that focuses on the main features that the application has to offer. Of course, we are going to include the synchronization's feature as well as some cases of conflict and its resolution.

6.2 System Usage

The most common action that takes place in the application is for someone to add a new pet in the system. As you can see in the following figures, the main screen asks for the user to add some more important information regarding the new pet. Some of this important information includes the name of the pet, its breed and species, its gender and a profile photo if the user desires to add one. In addition to this, a whole bunch of additional information regarding someone's pet can be added through the advanced mode feature. Information about the registration data of the pet, its parents and much more. When the user verifies the pet's addition, the application recommends the user some reminders about the pet's care (bath, ear/nails/teeth/hair cleaning etc.) based on the usual habits of the pet's species.

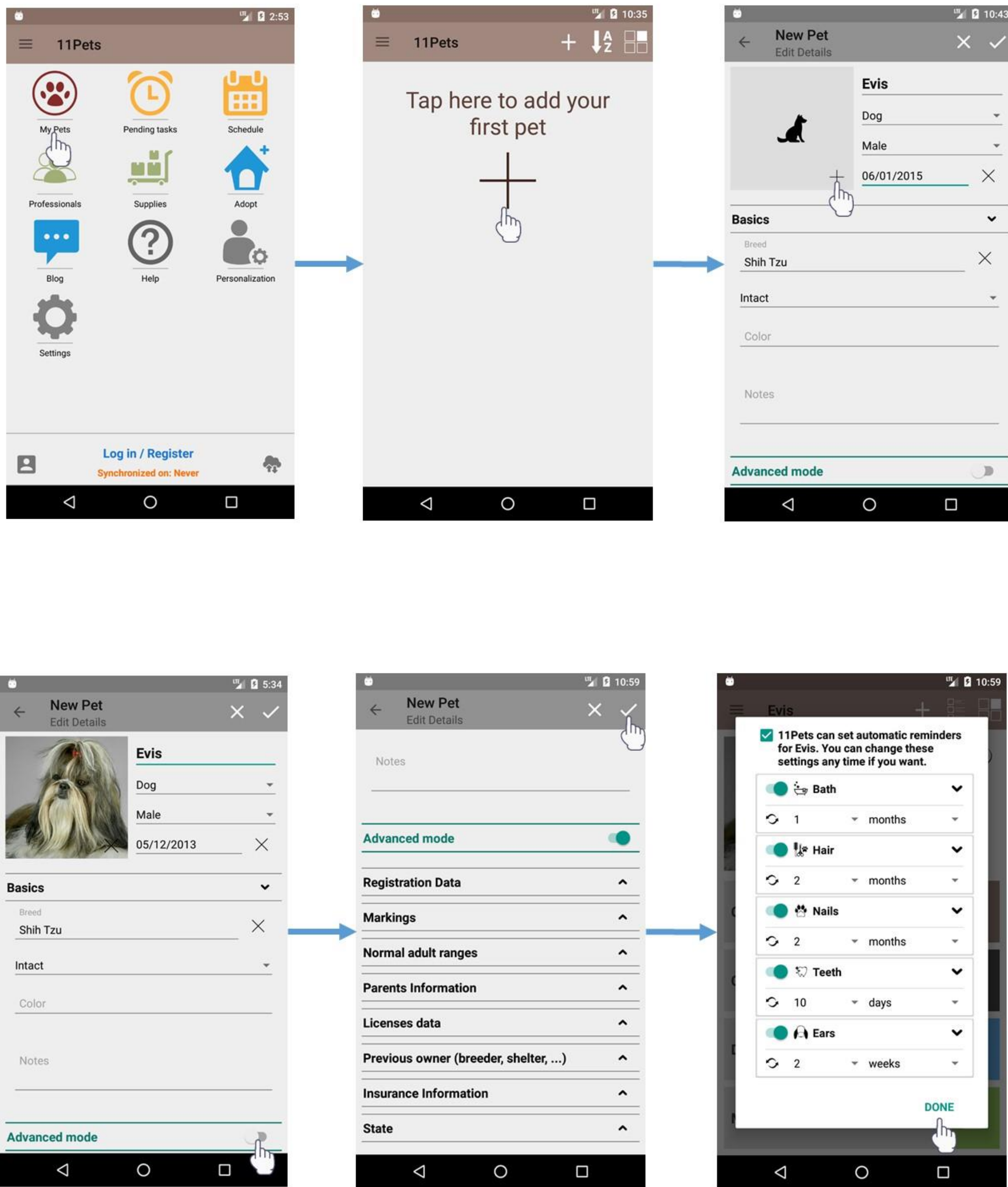


Figure 22 - Add Pet

After the pet is added, through the screen of the pet's details someone can choose one of the four main categories of information each pet has. These categories are the following:

Gallery Category – Figure 23

From here, the user can add photos or notes about the daily life of the pet and create a timeline of all these events.

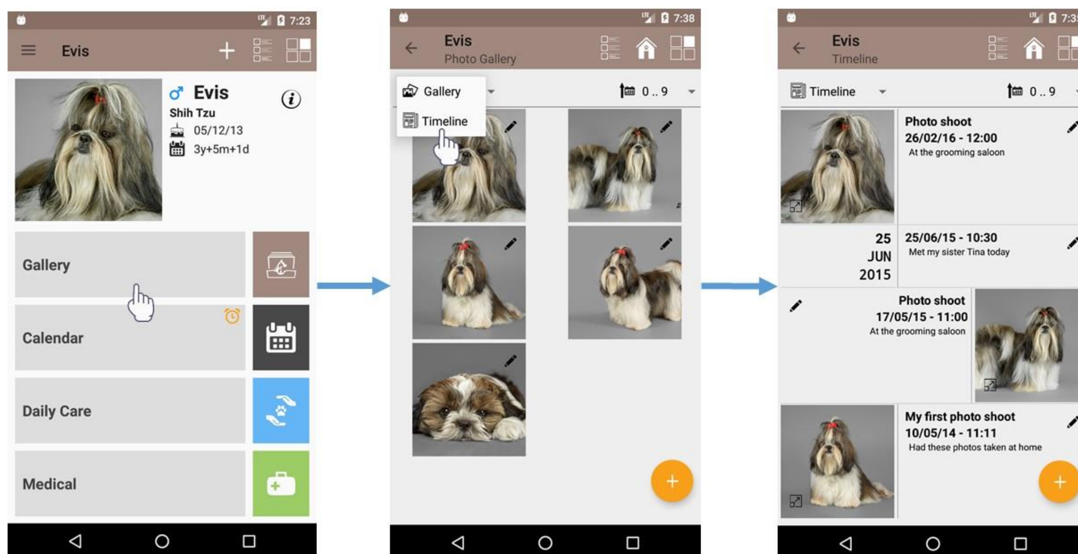


Figure 23 - Pet Gallery

Calendar Category – Figure 24

All the reminders that concern the pet's food, medication, supplies and daily care events show up in here. The user can click on an event to see more details of it or mark it as done, choose to skip it or snooze its reminder to fire up at a later time.

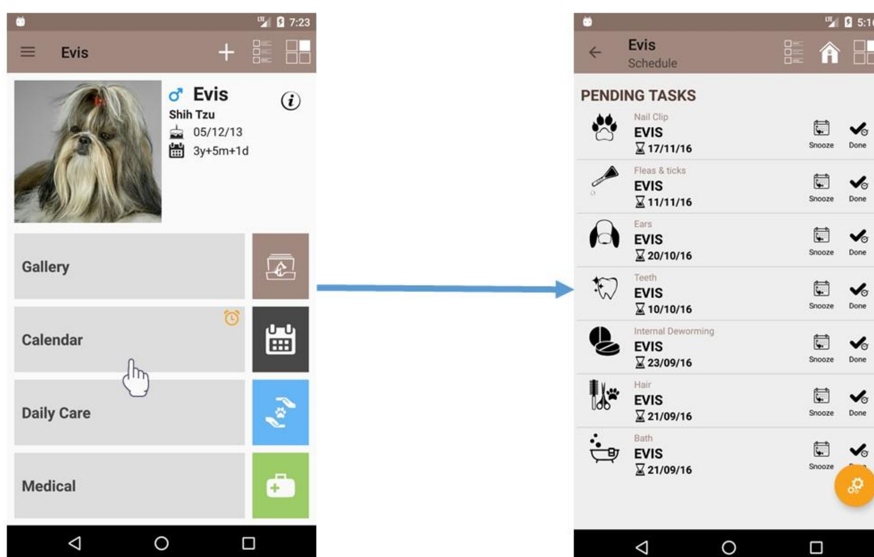


Figure 24 - Pet Schedule

Daily Care Category – Figure 25

The pet's foods, medication vaccines and daily care information goes in here as well as measurements and many more.

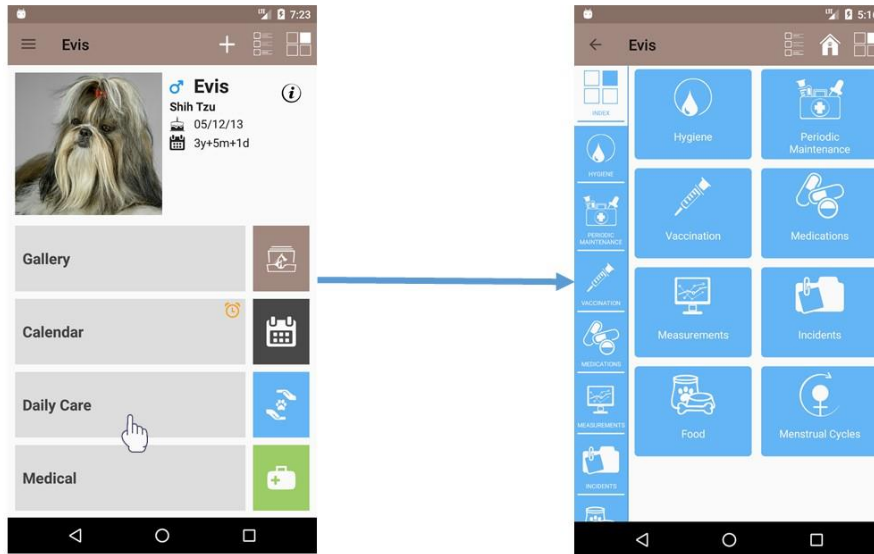


Figure 25 - Pet Daily Care

Medical Category – Figure 26

In here, information like lab and genetic test results can be saved as well as the pet's medical conditions, allergies and any surgeries the pet might have gone through.

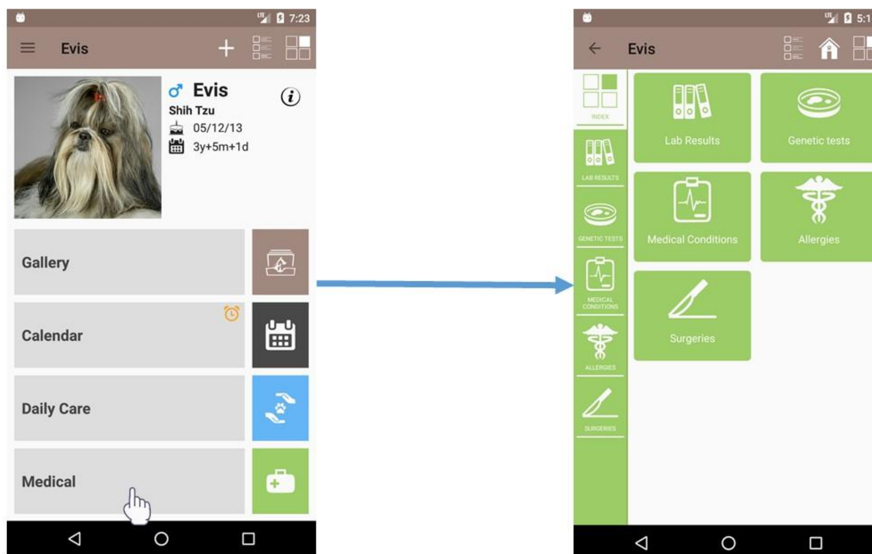


Figure 26 - Pet Medical

Moreover, the 11Pets company maintains a web platform for Pet Adoptions. On this platform, various shelters or pet rescue centres can register and then login and create a page for each of the pets that are available for adoption on their institution. The android application is connected with this platform and all these pets that are available for adoption can be virtually visited from the application as well. The interface of the android application regarding the adopt is shown in Figure 27.

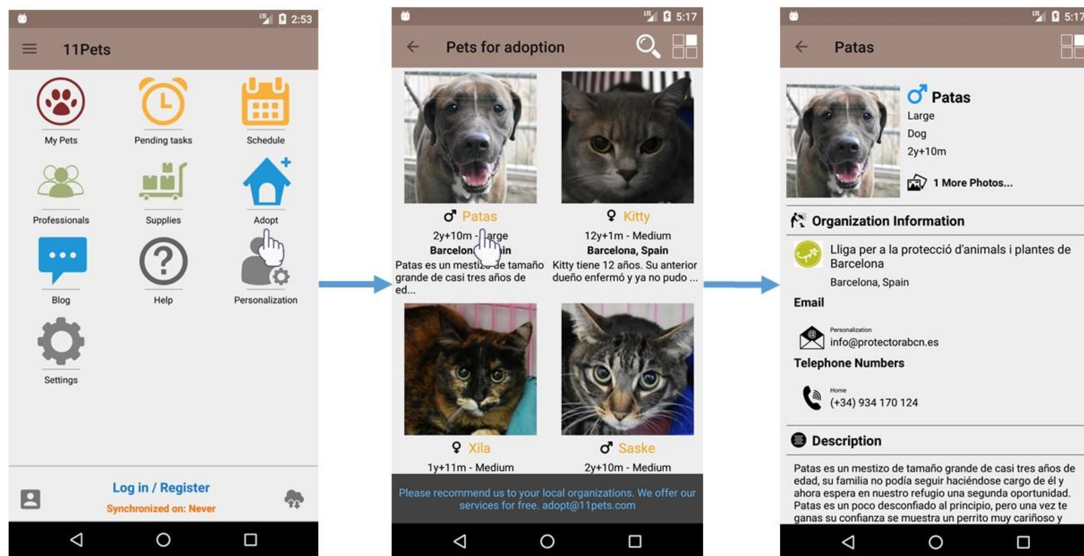


Figure 27 - Adopt

Another feature the 11Pets android application provide to the users is the Professionals feature as shown in Figure 28. From here, the users of the application have the opportunity to search for various types of professionals that are registered in the Google Maps Services [8]. The search can be done using either the current location of the user of any other location of its choice. Also, there are many types of professionals to search for, like vets, breeders, trainers and more or even search for any other type of professional the user wants to.

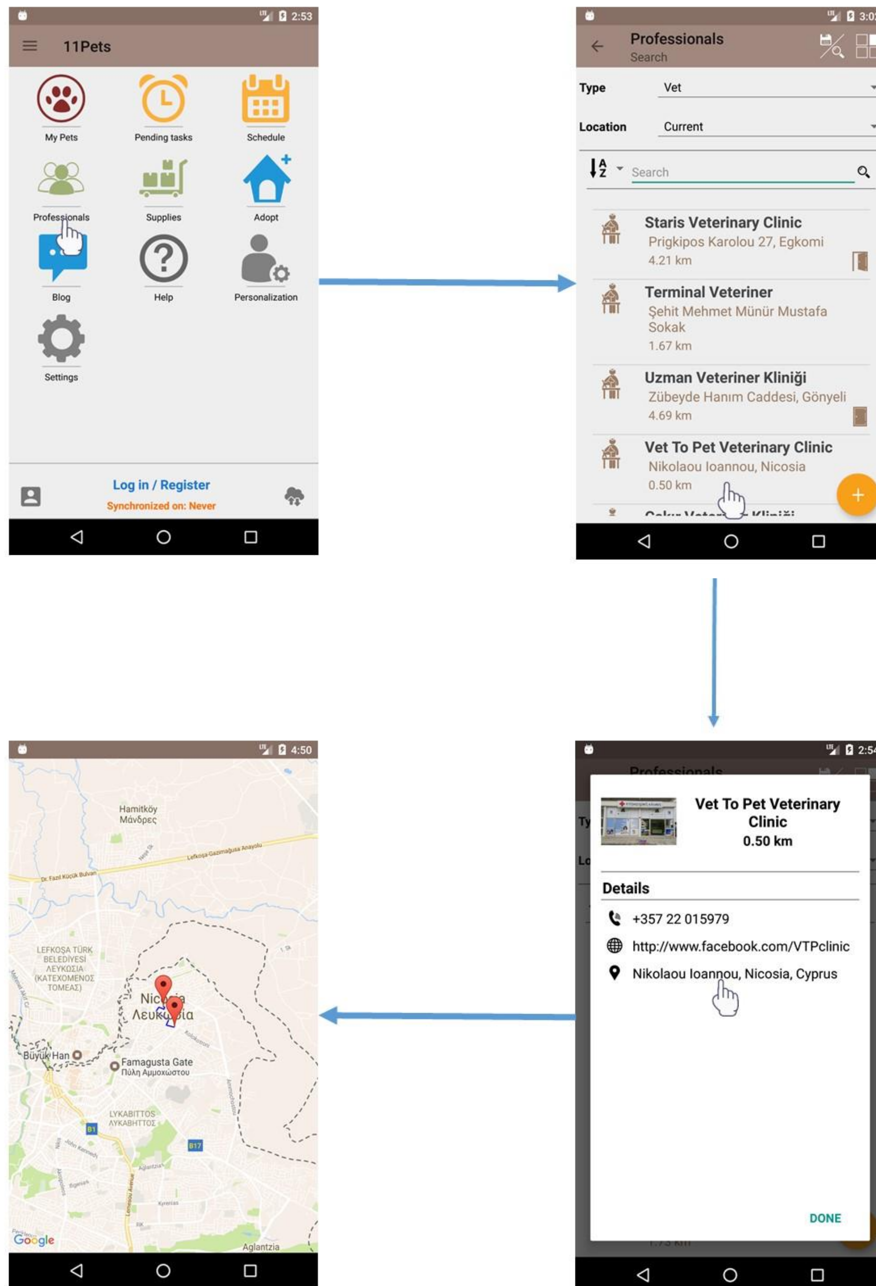


Figure 28 - Professionals

What is more, there are the features that concern this Individual Diploma Thesis, the synchronization algorithm. To start with, in order to be able to use the synchronization's feature, one must register to the 11Pets community first. Figure 29 shows how the registration can be done from the screens of the application.

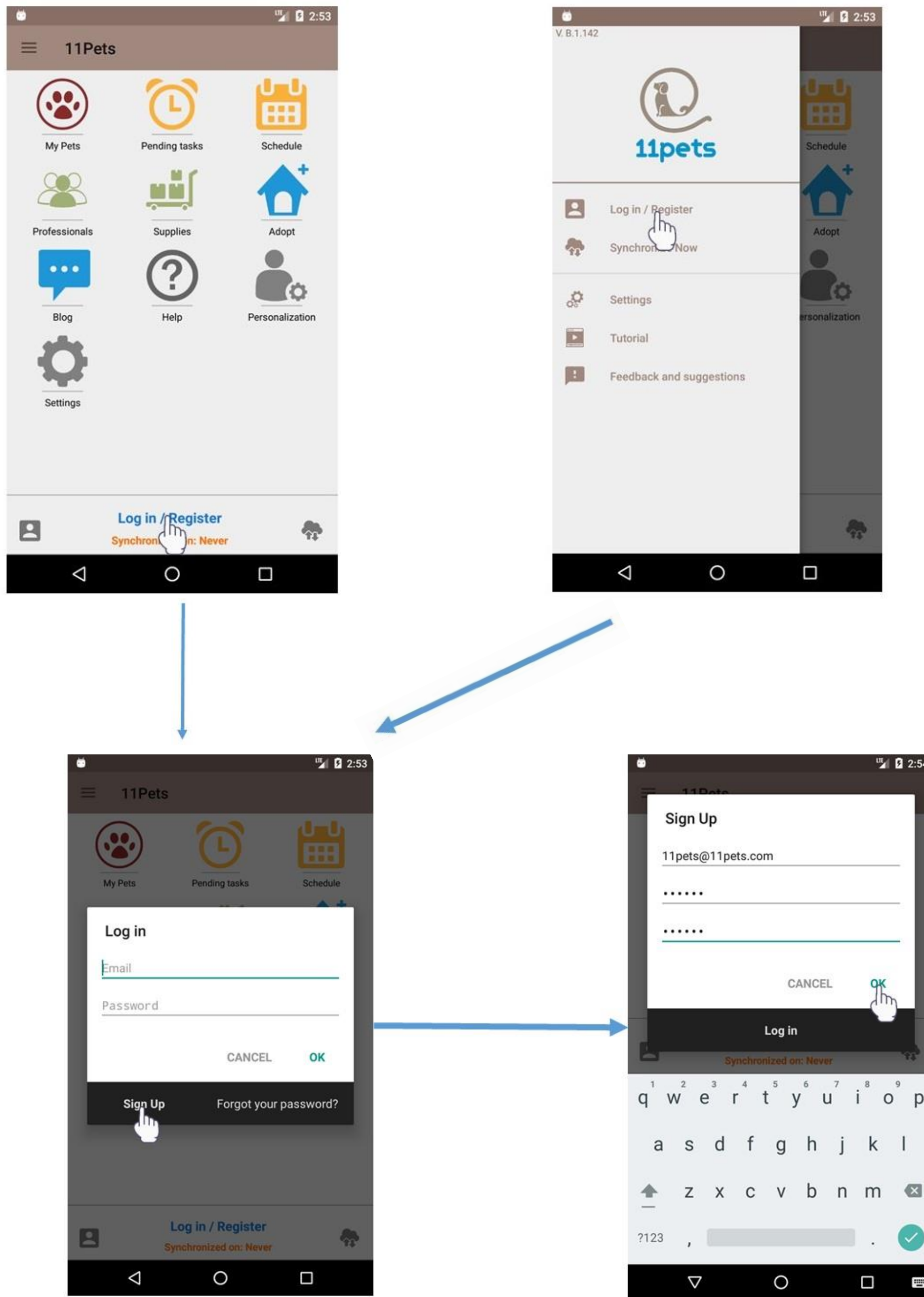


Figure 29 - Register

As was previously stated, the 11Pets application used to give its users the opportunity to backup and restore their data, to and from the server, in order to keep them safe and replicated. With the start of the synchronization era, these requests won't be used anymore. In any case, we came up with a neat and smart way to keep the restore feature still available but hidden from the eyes of the users. The restore feature might be needed in case of a user that wants to restore its data but no synchronization runs were made before. This means that the data the user is looking for can only be obtained from the restore feature. The secret restore becomes accessible for use if the user holds the sync icon for five seconds as can be seen in Figure 30.

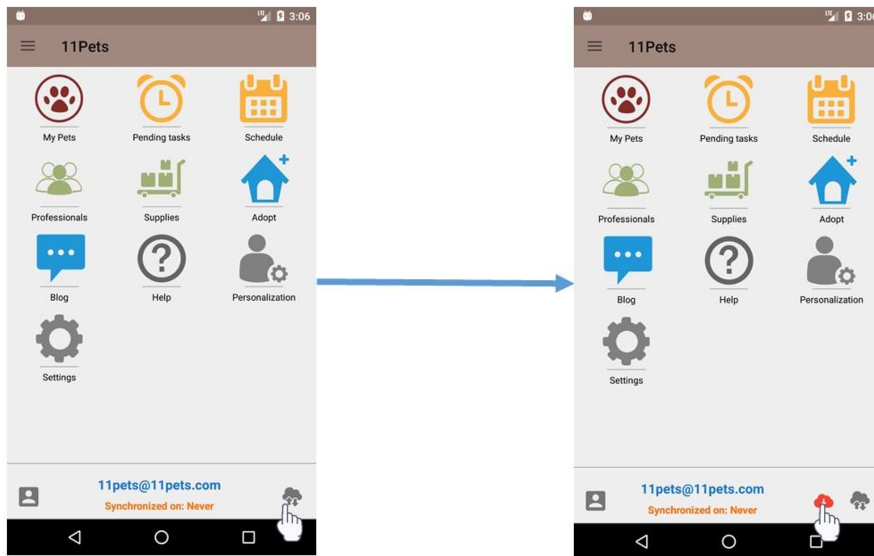


Figure 30 - Restore

To return to the subject of synchronizing, there are two ways to run the synchronization cycle. The first one is done on the foreground and with the user's command and the other one is done in background whenever it is called as necessary by the application itself. The user can manually launch the synchronization algorithm as shown in the following figure.

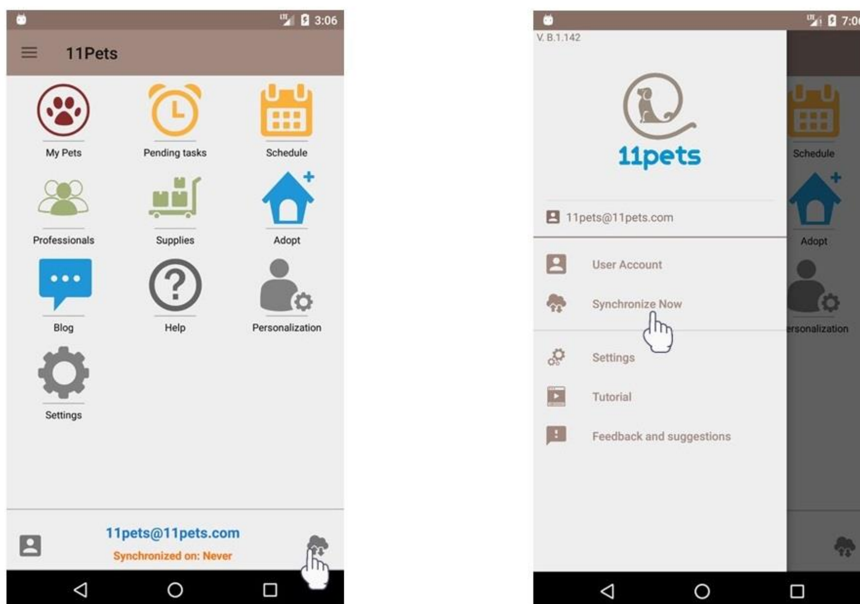


Figure 31 - Synchronize

The following figures show the progress bar that is shown to the user while the synchronization is running. The progress bar informs the user on which of the four phases of the algorithm is currently running and the total progress of the algorithm.

Phase A – Setting Up – Figure 32

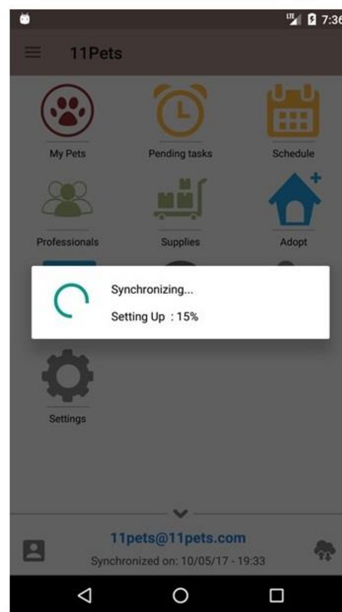


Figure 32 - Setting Up

Phase B – Processing Data – Figure 33

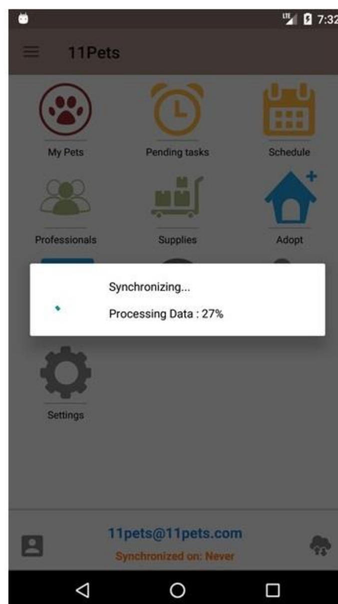


Figure 33 - Processing Data

Phase C – Downloading Files – Figure 34

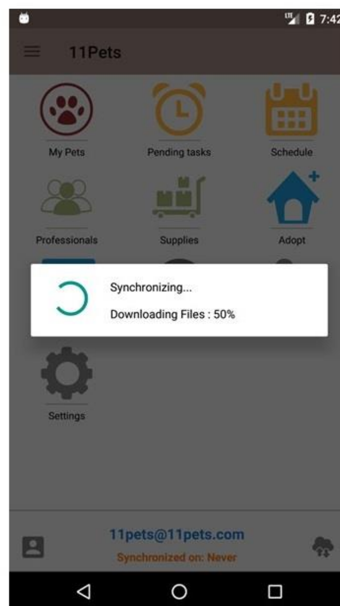


Figure 34 - Downloading Files

Phase D – Uploading Files – Figure 35

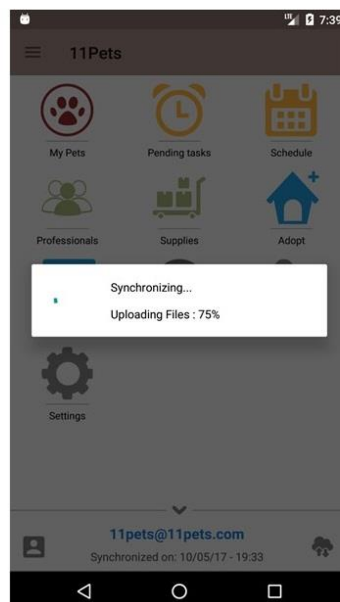


Figure 35 - Uploading Files

Finally, if the synchronization results in a conflict then an icon appears on the toolbar of some of the screens of the 11Pets application, the most commonly used screens. As it can be seen on the following figure, with the user's click on the icon the conflict resolution activity is launched and the user is called to choose how to resolve the conflict that has occurred.

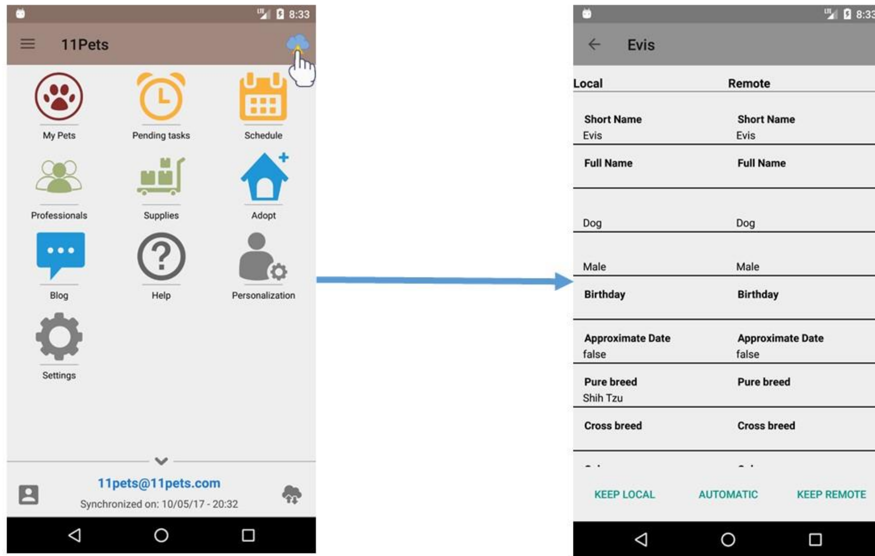


Figure 36 - Conflict Resolving

Chapter 7

Empirical Evaluation

7.1 Metrics Overview 57
7.2 Metrics Evaluation 58

7.1 Metrics Overview

An important phase in the lifecycle of a system it's its evaluation. The task of evaluating a system comes after the implementation is completed. The metrics obtained from the evaluation can give the developer a better understating of the performance of the system and help find various optimizations and improvements for the system. Regarding our system, we will be focusing on the time needed for the main phases of the algorithm and the most common transactions. At the meantime, the proper infrastructure was used to provide us with metrics about the breakdown of the algorithm. This means that we have metrics that concern each phase of the algorithm which will allow us to go even deeper and focus our efforts on specific transactions and processes.

For our metrics, he used two individual devices. The first device was a Samsung Galaxy A5 (2017) and the second device was a Huawei Nexus 6P. The table that follows overviews the specifications of the two devices.

	Samsung Galaxy A5(2017)	Huawei Nexus 6P
OS	Android 6.0.1 (Marshmallow)	Android 6.0 (Marshmallow)
CPU	Octa-core 1.9 GHz Cortex-A53	Octa-core (4x1.55 GHz Cortex-A53 & 4x2.0 GHz Cortex-A57)
RAM	3 GB	3 GB

For this first evaluation effort, we were able to obtain the following metrics:

- Synchronization of an empty database for the very first time
- Synchronization without any changes
- Synchronization with one new entry
- Synchronization with one updated entry

7.1 Metrics Evaluation

What follows, are the metrics taken from the real time runs of the algorithm on the two devices mentioned above. The metrics shown are an average of about 25 observations each.

Synchronization of an empty database for the very first time

Average = 41.4 seconds on Samsung A5 and 44.8 on Nexus 6P. In Figure 37, this metric shows as the average of the time needed to a new user to synchronize for the first time. The time taken is acceptable because all of the *seeded entries* are to be synchronized for the one and only time. These entries are approximately one thousand (1000). The synchronization for the very first-time metric will be extended in order to see the time needed for not empty databases to sync for their very first time.

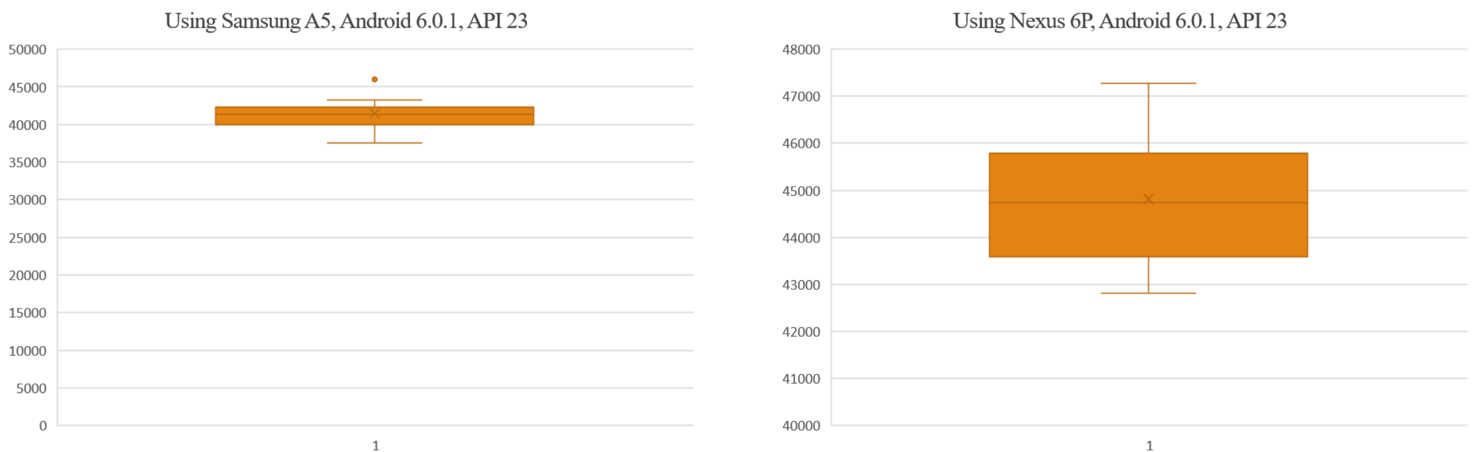
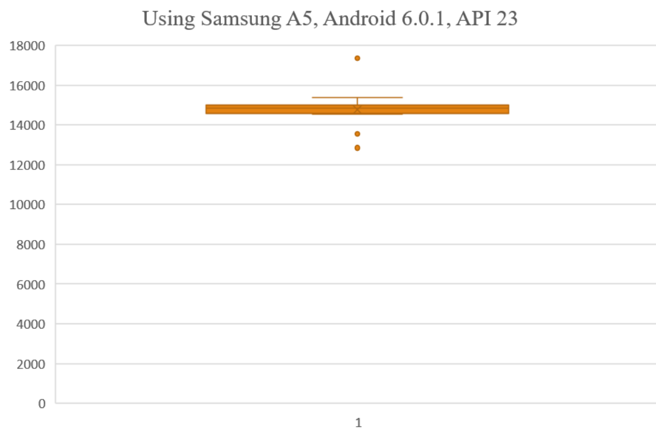


Figure 37 - Synchronizing an empty Database for the first time

Synchronization without any changes

Average = 14.8 seconds on Samsung A5 and 13.4 on Nexus 6P. This is the simplest scenario of synchronization. As we can see in Figure 38, the average time needed is very low and we are very pleased for this. This is also a common scenario because the application gives its users the opportunity to the synchronization to be launched whenever the application starts. A future metric regarding the no changes synchronization could be the times this scenario actually occurs compared to other synchronization scenarios.



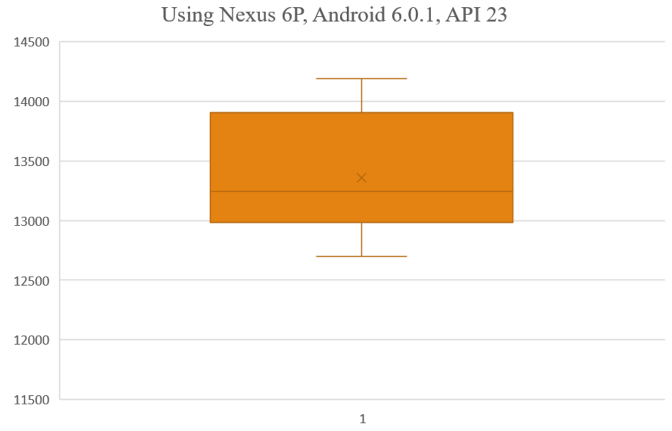
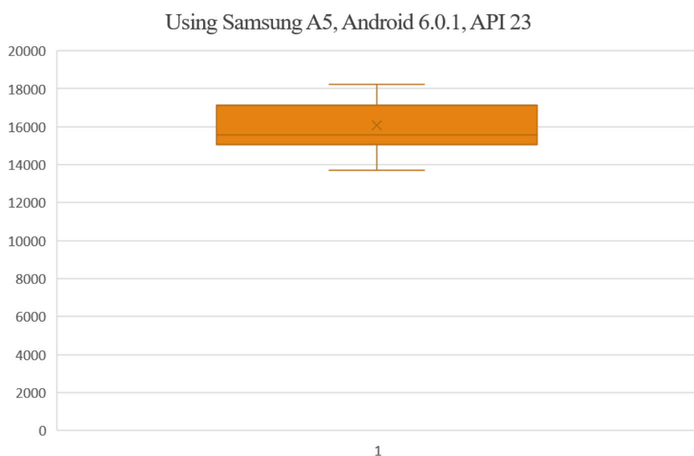


Figure 38 - Synchronizing without any changes

Synchronization with one new entry

Average = 16.1 seconds on Samsung A5 and 15.2 on Nexus 6P. The one new entry synchronization is a scenario that may occur very often due to the fact that synchronization launches after a modification is done on the device's database. As shown in the following figure, the average time differs from the no changes synchronization by about two (2) seconds. This tells us that an addition of a new entry will only increase our sync time by an average of two (2) seconds. The most important is that no difference was noticed when adding entries on different tables. This shows stability of our server and extensibility on new tables addition.



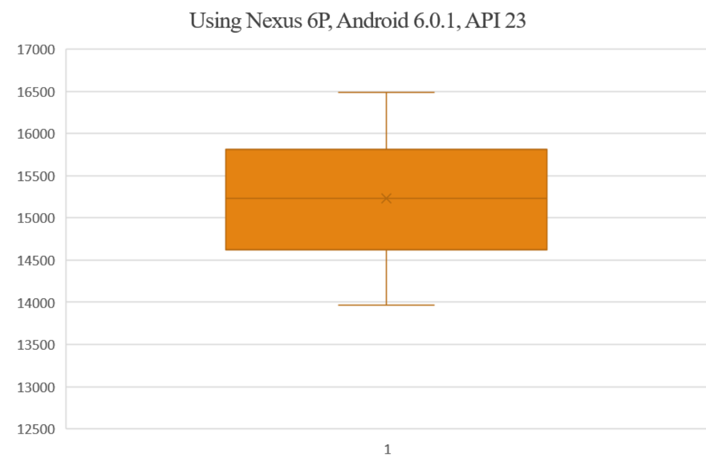


Figure 39 - Synchronizing with one new entry

Synchronization with one updated entry

Average = 15.2 seconds on Samsung A5 and 14.3 on Nexus 6P. Just like the new entry synchronization, the updated entry synchronization is a common scenario due to the launching decisions we have taken for our algorithm. In figure 40, we can see that the time needed for this scenario compared to the no changes synchronization is higher by an average of 1 second and lower than the new entry scenario by an average of 1 second. Again, just like the new entry synchronization, the update time on different tables showed no difference between them.

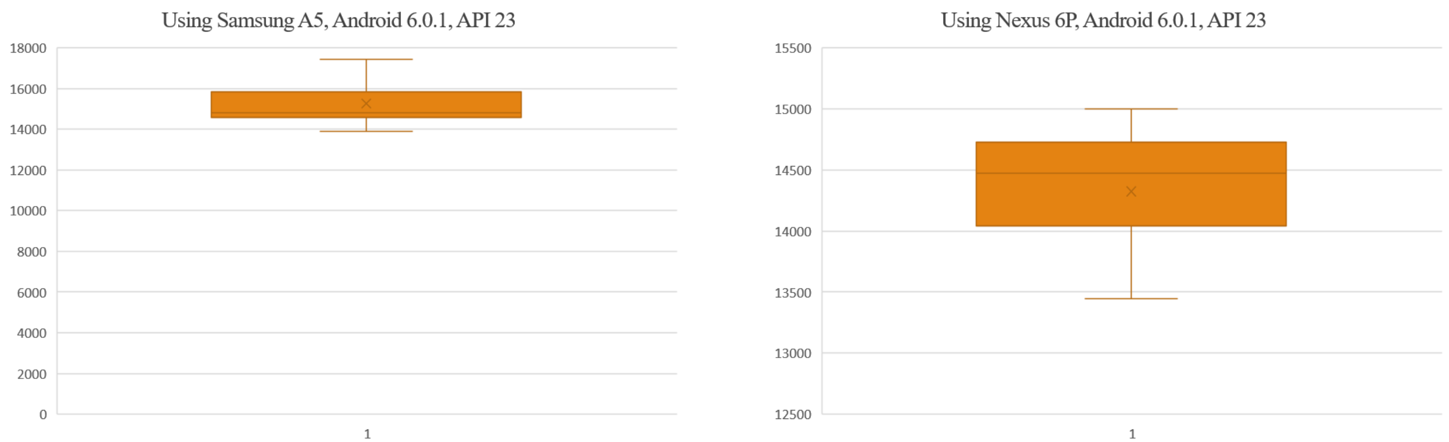


Figure 40 - Synchronizing with one updated entry

The most important conclusion that we were able to exclude from our metrics is the consistency of the requests. All of our observations are very close to the average and no important outliers came up. This means that the server is very trusty and responds to request with fairness. Also, what we could understand is that there are no differences on times synchronizing different tables. The flow charts of the updated and new entries presented above include numerous tables and the times needed were the same. As a future work, the breakdown of the algorithm will be evaluated in order to find the best optimizations possible to improve our algorithm.

Chapter 8

Conclusions and Future Work

8.1 Summary 61
8.2 Future Work 62

8.1 Summary

As a final chapter, we shall take a moment to rethink all of the work we have done and set our goals for what's yet to come. We have started our work by getting a better understanding of what we are actually trying to achieve. We got to know better the inner structure and needs of the Startup Company and its users. An algorithm on a piece of paper was presented to us, alongside a set of new concepts that were basically unknown to us. The study and analysis of this algorithm resulted in a brainstorming of ideas and questions that needed to become reality. A fair amount of time was consumed in the general study of similar concepts and problems in order to get a more spherical idea and knowledge of what problems we have to face. Some concepts are the Synchronization, the data Replication and the Conflict Resolution.

Eventually, we were ready to start designing our flow charts and writing code down. First of all, the Entities, DAOs and DTOs were implemented in their final form. All these classes contained a lot of methods that had to be carefully designed because even a single mistake would lead our data to become corrupted. Next, we implemented a Unit Test class for each of our database's tables and validated that all of the above structures worked as expected. Moving on, the Apache to Volley transformation took place. A very important and time-consuming task that gave the Startup Company the ability to move on with the evolvement of the application. Having these implemented we were ready to create our Synchronization class and merge all of these features together to complete the algorithm. The four phases of the algorithm were presented and implemented with great caution. Specifically, we implemented the Setup, Process Server Changes, Process Device Changes and Process Files phases. From this moment and on, we were able to run tests on our

algorithm whenever we wanted. Before proceeding with tests and debugging though, we had to design the conflict resolution mechanism. This mechanism, is consisted by a numerous of class objects alongside an activity that is responsible to provide the user with the graphic interface and handle its choice. To return to the subject of synchronizing, we initiated a series of debugging phases that would help us spot any mistakes done or any bugs left behind. The first phase was done locally inside the Company's boundaries with some fake data. That was helpful, but we are aware that live databases are not so clean and neat as a new one created for debugging. Seeing that, the second phase included a silent synchronization release. I mean, that the users were aware of an inter-company algorithm running on their devices but without them knowing what it was doing and without seeing any results or actions to take place. This is due to the fact that the algorithm was running on the background of the application. Finally, we performed an experimental evaluation to obtain a better understanding of the latency of the main system transactions.

8.2 Future Work

Although a large amount of time and recourses were needed so far, the 11Pets company's plans for the synchronization algorithm haven't finished. It is no doubt that an in-depth analysis of the metrics of the algorithm is needed. Namely, for every individual request and for each of our database's tables, the time of the communication between the application and the server must be reported. Using all of these metrics, someone could find slow requests, requests that carry a lot of information and could be split and so on. Moreover, a very interesting study is the launching moments of the synchronization algorithm. Should the synchronization algorithm run whenever a change is made? Should it run after x changes? What if an entry changed contains essential information that must be propagated as soon as possible to the other devices? Similarly, one may come up with a lot of great ideas and questions that are interesting to consider and study. In parallel, the limits of the server in terms of scalability and high contention will be tested in order to see whether it can respond to great level of concurrent transactions. All the above thoughts, hide another large project and set of concepts that the 11Pets is asked to conquer.

In the meantime, the 11Pets has widened its services and an iOS application is being developed. Consequently, all of the material studied and implemented in this thesis will be carried out on the iOS environment as well. As a matter of fact, all of the research and design of the algorithm with its supporting structures are ready as a result of this work. What will be needed is an initial study of the iOS environment and its capabilities. The Volley and the Unit

Testing are features that may differ from the android ones that we have implemented. When this is accomplished, all of the ideas, the algorithm itself and the issues that may appear have already been studied and all its left to do is the code transformation.

All in all, the synchronization algorithm has great potential to improve and become even more efficient. The optimizations based on the metrics will result in an even faster and more efficient algorithm. Finally, the iOS synchronization algorithm will potentially give us more feedback about our algorithm's efficiency and allow users that work on the iOS environment to benefit from the 11Pets service as well.

Bibliography

[1] Andrew S.Tanenbaum and Maarten Van Steen, "Distributed Systems, Principles and Paradigms", 2nd Edition, 2007, Cited on pages 274-275, 303.

[2] Hu Jieping and Yang Shulin, "Research and implementation of Web Services in Android network communication framework Volley", IEEE-ICCSN, June 2014.

[3] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer and Brent B, Welch, "Session Guarantees for Weakly Consistent Replicated Data", In Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems (PDIS 94), IEEE-ICCSN, 1994, pp. 140-149.

[4] Jonathan P. Munson and Prasun Dewa, "Sync: A Java Framework for Mobile Collaborative Applications", Special Issue on Executable Content in Java, IEEE Computer, 1997, pp. 59-66.

[5] Cristiana Amza, Alan L. Cox and Willy Zwaenepoel, "Distributed Versioning: Consistent Replication for Scaling Back-end Databases of Dynamic Content Web Sites", In Proceedings of Middleware 03, pp.282-304.

[6] Wolfgang Gatterbauer and Dan Suciu, "Data Conflict Resolution Using Trust Mappings", In Proceedings of the 2010 ACD SIGMOD International Conference on Management of data, pp. 219-230.

[7] Philip A. Bernstein and Nathan Goodman, "Concurrency Control in Distributed Database Systems", ACM Computing Surveys (CSUR), Volume 1 Issue 2, June 1981, pp. 185-221.

[8] Manav Singhal and Anupam Shukla, "Implementation of Location based Services in Android using GPS and Web Services", IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 1, January 2012.

[9] Israel J. Mojica Ruiz, Meiyappan Nagappan, Bran Adams and Ahmed E.Hassan, "Understanding Reuse in the Android Market", ICPC, 2012, pp. 113-122.

[10] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek and Angelos Stavrou, "A whitebox approach for automated security testing of Android applications on the cloud", In Proceedings of the 7th International Workshop on Automation of Software Test, June 2012, pp. 22-28.

[11] Zach McCormick and Douglas C. Schmidt, "Data Synchronization Patterns in Mobile Application Design", In Proceedings of the Pattern Languages of Programs (PLoP) 2012 conference, October 19-21, Tucson, Arizona.

[12] Paul Pocatilu, Catalin Boja and Cristian Ciurea, "Syncing Mobile Applications with Cloud Storage Services", Informatica Economica, vol.17, no.2/2013.

[13] https://www.11pets.com