Decentralising a Mobile Application

Yann Gabbud

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Bachelor Project

June 2019

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Jeffrey Richard Allen
EPFL / DEDIS

Contents

1 Introduction of the subject and the problem.

1.1 Motivation and goal

The goal of this project is to identify a category of application where decentralization can be beneficial and to adapt an android application to operate in a decentralized context. The fist step consists to identify the parts of the app where data is stored, and then find a way to store that data in Cothority instead.

The motivations of this project are to see if it is possible to achieve such a task using Cothority, Calypso and Byzcoin and to see what are the limitations.

1.2 Choice and description of the app

Many applications are eligible for decentralization, such as note-taking, password managers or messaging. The important point is that the app stores data somewhere, either locally or on a server. At the beginning I wanted to choose an application similar to Dropbox but more simple and to substitute the default server by a cothority roster. However after investigation, I quickly came to the conclusion that this would be too difficult and that I will not have enough time given the complexity of the communication protocols and the lack of documentation.

So I decided to start from a simple note block application named Memento and to modify it to use it in a multi-user context to share documents. Here is a small description of the app. Many user can use the app to share text documents. Each user has one or many roles who defines his rights. There is three types of role, publisher, reader and delegated reader. A publisher can create and share documents with other users. He can also revoke at any time the read access of an user. A reader can read documents he has read access to and give read access to another user. This process is known as delegation.

2 Corresponding background

2.1 Overview

Nowadays, mobile apps store data centrally. It may be somewhere on the phone's memory or on a remote server from a large company. This centralization of data cause problems. In the case where the data are stored on the phone memory, if the phone is lost or a material failure happens, the data are definitively lost and this is not what we want. In the case where the data is stored on a remote server, it is necessary to trust the entity that owns the servers. We do not want this data to be used for commercial purposes, to be modified or to no longer be able to access it. We must therefore find

another solution to ensure both the conservation and the security of these data.

2.2 Blockchain and ByzCoin

These last years, we saw the rise of the blockchain. Here is its wikipedia's definition:

"By design, a blockchain is resistant to modification of the data. It is "an open, distributed ledger that can record transactions between two parties efficiently and in a verifiable and permanent way". For use as a distributed ledger, a blockchain is typically managed by a peer-to-peer network collectively adhering to a protocol for inter-node communication and validating new blocks. Once recorded, the data in any given block cannot be altered retroactively without alteration of all subsequent blocks, which requires consensus of the network majority. Although blockchain records are not unalterable, blockchains may be considered secure by design and exemplify a distributed computing system with high Byzantine fault tolerance. Decentralized consensus has therefore been claimed with a blockchain."[1]

We see that the blockchain responds well to the problem mentioned above. As the system is distributed, there is not a single point of failure. The system is not managed by a single entity but by a group that acts collectively. The data is encrypted and all alterations are visible. This is definitely what we need. In this project we use ByzCoin, a novel Byzantine consensus protocol that leverages scalable collective signing to commit Bitcoin transactions irreversibly within seconds[2].

2.3 Calypso and Darc

Blockchain is good but we need a way to store and retrieve document on it. For that purpose we use Calypso. Calypso allows to store symmetric keys in ByzCoin, protected by a sharded key, and controls access to this symmetric keys using Darcs, Distributed Access Rights Control. It implements both the access-control cothority and the secret-management[3].

3 Design and implementation

This section explain how the app works in a decentralized context. From here, it is assumed that the reader is familiar with Cothority, ByzCoin, Calypso and Darc. Also note that for the sake of simplicity the current implementation contains only three users, each with a different role. There

[1] https://en.wikipedia.org/wiki/Blockchain
[2] https://dedis.epfl.ch/page-135477-en-html/
[3] https://github.com/dedis/cothority/blob/master/calypso/README.md

is a publisher, a reader and a delegated reader. But the system can have as many user as needed and every user can have every role.

3.1 Server side and initial configuration

The server side is very simple. It consists of a roster composed of three conodes on which an instance of ByzCoin works.

The first step of the configuration is to create signers and to associate them with the different users of the system so that Cothority can uniquely identify them.

Then the genesis Darc used to configure ByzCoin is created using the makeGenesisDarc method. A first rule is added to this Darc so that the admin can spawn a value instance for each user. The usage of these value instances will be explained in the next sections. Note that the admin is not a user, it is a signer used only to create the value instances of the users so that users do not have this right. Then two rules per user is added to the genesis Darc. The first rule allows a user to spawn Darcs and the second rule allows him to evolve a value instance.

3.2 General design

A private key pair and a value instance is associated to each user. The private key pair is not the public and private key of the signer but a new set of public and private keys used to re-encrypt data when spawning a read instance. The value instances are used as a asynchronous communication medium between two users. When a user want to send data to another user, he simply write these data into the value instance of the user.

Since the application works in a decentralized context, no documents are stored on the memory. Only information to retrieve documents on ByzCoin is stored on the memory. Therefore, when a user starts the application, the first step is to retrieve the documents on ByzCoin and also to check his value instance to see if new documents have been shared with him.

Once this step is done, the user has access to his documents. He can now, if he wishes, create a new document that will be saved on ByzCoin and share it with other users. To do this, he must give the right to read to the users with whom he wishes to share the document and sent them the necessary information so that they can retrieve the document on ByzCoin.

Finally, a user can share a document that has been shared with him. To do so, he must modify the access rights to this document and send the necessary information so that the user with whom he wishes to share the document can retrieve the document on ByzCoin. This action is called the delegation.

3.3 Creation and sharing of a new document

The first step is to create Darcs, the publisher Darc and the reader Darcs.

The publisher Darc defines the rights of the users on the document. It contains the following rules: the publisher can spawn a write instance, spawn a read instance and evolve the publisher Darc. The last rules state that the signers of the reader Darcs can spawn read instances of this document. The publisher is the only owner and the only signer of this darc. Therefore, he also has the possibility to sign and evolve the darc.

The reader Darcs specifie who the publisher shares the document with. Note that if the publisher wants to share his document with multiple users, a reader Darc per user must be created. The reader Darcs contains only the default rules which states that the reader can evolve the reader Darc and sign. Later new signer will be added to this Darc by the reader during the delegation process to allow them to read the document.

Once the Darcs are created, the publisher spawns a write instance. To do this he uses the newly created publisher Darc and provides the document to be published on ByzCoin. Note that the document is not provided as is, but it is parceled beforehand to become a byte array.

After the write instance is created, the publisher spawns a read instance so that he can later get back the document from ByzCoin. In this step he provides the public key generated during the configuration as re-encryption key.

Now that the document has been created and stored on ByzCoin, the necessary information to manage and retrieve the document have to be save somewhere. What has to be saved is the ids of the Darcs, the write instance and the read instance. Because these informations are not sensitive, they are stored in clear in the memory. It is explained in the following subsection where and how these data are stored.

Finally, the last step is to transmit the informations necessary to access the document to the readers. This informations are the ids of the reader Darc and the write instance. To do this, the publisher writes to the value instance of the readers.

3.3.1 Parcelization

The parcelization is a process of serialization. This process is similar to the well known serialization of Java. It is used for marshaling and unmarshaling Java objects but is more efficient and only usable for android. The drawback is that more work is left to the programmer. In this project, parcelization is used to transform a document composed of an id, a creation date, a title, and a body into a list of bytes that can be put in a Calypso document and stored on ByzCoin. Parcelization is important because only an array of byte can be stored into a Calypso document.

Unparcelization is the inverse process. A byte array is converted into a document.

3.3.2 Storage of the instance ids

In order to be able to retrieve and manage documents stored on ByzCoin, it is imperative to save the write instance id, the read instance id and the darc instance id of each document. To solve this problem, the shared preference system of android is used. This system allows to save and retrieve data in the form of key-value pair. The ids of each document are therefore stored in the shared preferences

3.3.3 Value instance as message passing

The communication between two users is done through value instances. When a user wants to share a document with another, it is enough for him to write the necessary data in the value instance of the other user. Since data can already be present in the value instance, the new data is append to the old data. In this way we are sure that there is no data loss. Once a user retrieves the data of his value instance he cleans it to avoid retrieving the same data twice.

Because a value instance only allows a byte array to be stored, the data stored on the instance value is structured as follows. Each id is separated by a semicolon and each set of ids of a document is separated by a comma. Once the data of the value instance are retrieved, it simply have to transform the array of byte into a string and to parse it according to the the commas then parse again the sub-string according to the semicolon to get all the ids. You can see here a representation of the data in the value instance.

```
...readerDarcID:writeInstanceID,readerDarcID:writeInstanceID...
```

This communication process is very simple and repose on Cothority which is a good point. However it has several flaws that we will be explained in the evaluation section

3.4 Fetching documents on ByzCoin

This part has two stages. The first stage retrieves documents whose ids are known and stored in memory. First, the memory is read. Then the Calypso document is retrieve from ByzCoin by providing the read instance id as well as the private key needed to decrypt the document that has been re-encrypted with the public key of the user. Once the document is decrypted, it is unparceled and the first step is completed.

The second stage looks in the value instance if some documents have been shared with the user. If so, for each write instance id in the value

instance, the write instance is retrieved from ByzCoin and used with the user's public key to generate the read instance. Once the read instance is created, the document can be decrypted with the help of the private key of the user, and then unparceled. Finally, the ids stored in the instance value as well as the id of the read instances are saved in the memory for later reuse.

3.5 Delegation

The user selects a document that he wishes to share with another user. The write instance id and the reader Darc id of the document are retrieved from the memory. A secure Darc instance of the reader Darc is created. The reader Darc is modified so that the new user is now a signer of this Darc and therefore he can read the document. Finally the Darc is evolved to take into account the change.

The second step is similar to the process of sharing a new document. The ids are simply added in the value instance of the new user. Note that the user can select several documents and share them at once to another user.

3.6 Overview diagram

The diagram in figure 1 shows the process of creating a document up to its delegation. Points 1 to 5 are the actions performed by the publisher. Points 6 to 9 are the actions performed by the reader. Finally, points 10 to 11 are the actions performed by the delegated reader. Note that the value instances do not appear on the diagram and that the reader darc does not contains any delegated reader once the publisher creates it. The delegated readers are added to the darc by the reader in step 8 when he evolves the darc.

4 Evaluation and analysis of the results

4.1 Evaluation of the implementation

This sub-section is intended to evaluate the current implementation. The system works but has some flaws and weakness that will be addressed here

4.1.1 Value instance

The value instances are a very convenient way of communicating because they are super simple to use and they are based on Cothority. Therefore, it is not necessary to call an outside service to communicate. However, the modification of a value instance is not atomic which poses a big problem of concurrency. If two users want to share a document with the same user and

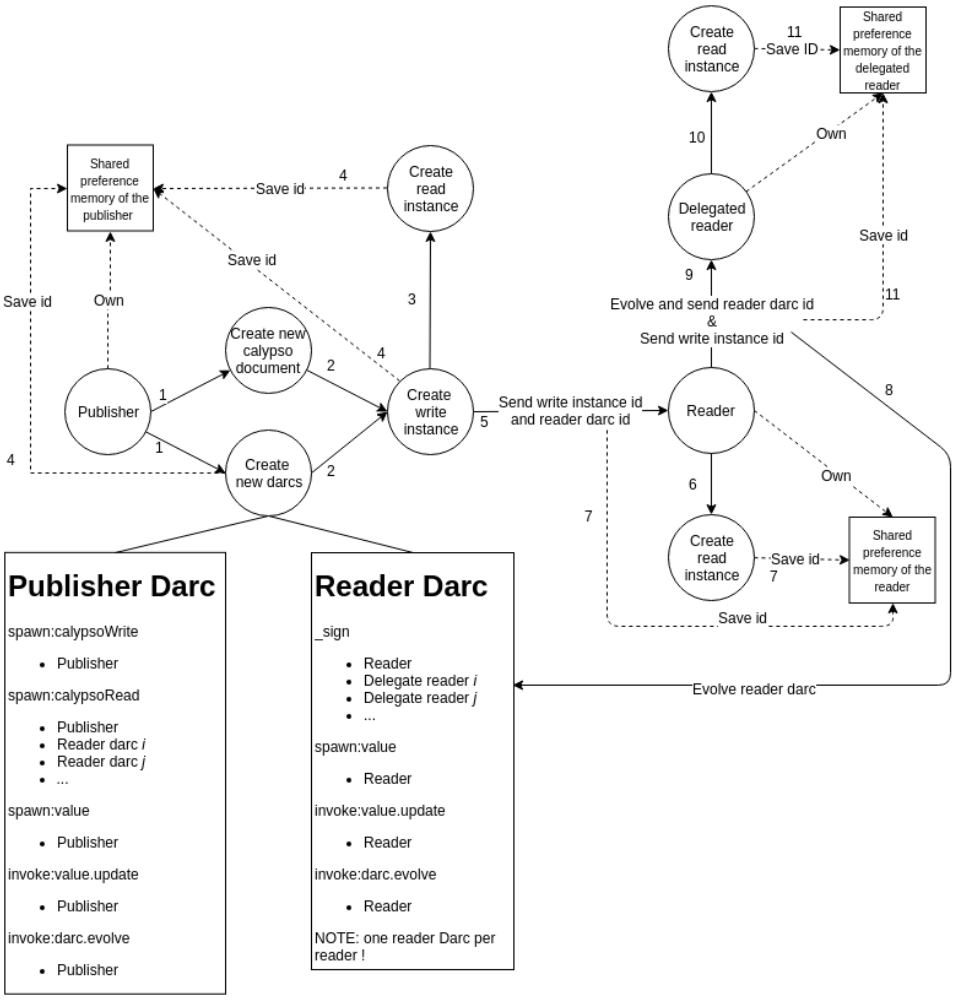

Figure 1: Memory profiling without pool

do so at the same time, the result can be unpredictable. The modification of one of the two users can overwrite the preceding modification of the other. Another case where the concurrency is problematic occurs when a user adds an element to the value instance while its owner is cleaning it. This element can be lost.

The concurrency problem occurs because it is necessary to recover the contents of the value instance before modifying it. Once the content is retrieved, it is modified (append or clean) and finally the change is published on Cothority. One possible solution to solve this problem is to add two atomic methods. The first must allow to append data in the value instance without having to retrieve the contents of the value instance. The new data is simply written at the end of the data already in the value instance. The second must allow to retrieve the data and to empty the value instance

atomically. The users who want to communicate use the first method and the user who receives the message uses the second method. In this way, there is no more concurrency problem.

Ideally, another mechanism of communication should be used because value instances have not been designed for communication purposes.

4.1.2 Inconsistency in the access right

When a user, the publisher, gives access to a file to a second user, the reader, this second user can in turn give access to the file to another user. The reader can give access to the publisher by including him in the reader Darc. Therefore the owner of the document appears both as the publisher and as a delegated reader. This conflict is problematic because Cothority does not know what rights to give to this user.

The library has been modified and the problem of infinite recursive darc delegation is now solved.

4.1.3 Avoid multiple delegation

In the current implementation with three users, the mechanism preventing multiple delegation of the same document to the same user works as follows. When the reader has delegated the document to the delegated reader, he notes in memory (shared preferences) that the document has been delegated and therefore he can not delegate it a second time.

However, this mechanism is not very practical because it involves storing data of the proper functioning of the system on the memory. In addition, with many users it would be necessary to store a list that would not be easy to maintain. In addition, this information is fully contained in the reader Darc so it is a duplicate.

A simpler method would be to check if the user with whom he want to share the document is already in the reader Darc. If this is the case, the delegation is canceled. Otherwise, it is done as usual.

4.2 Performance analysis

It should be known that the system performance is for the moment very bad because of a problem of garbage generated by the cryptographic functions of the library. This subection explains step by step what has been tried to improve the performance. Noted that initially the problem of garbage was not known.

4.2.1 Avoid doing unnecessary stuff

The first optimization was to test if a heavy operation is necessary before executing it. A heavy operation is an operation that communicates with

Cothority. For example, if the value instance is empty, the user will not tell Cothority to clean it. Thanks to this, a communication with Cothority is avoided. If the user is the publisher, he never receives communication. It is therefore not necessary to control his value instance when recovering its documents. Moreover if the space in memory where he stores the ids of his documents is empty, the app does not connect to ByzCoin. Here again unnecessary communications with Cothority are avoided.

This optimization is minor and does not bring any significant gain however I think they are necessary.

4.2.2 Connection to ByzCoin when launching the application

Whenever the application wants to publish or retrieve a document, modify a value instance or delegate access to a document, it must first establish a (re)connection to ByzCoin. The problem is that a connection to ByzCoin is quite slow. To avoid to do this action too many time, a change was made so that the application connects to ByzCoin at launch and maintains the connection as long as the user is using it. Subsequently, all the operations that need to use ByzCoin do not need to reconnect each time and therefore the performance is better.

4.3 optimization of a Calypso method

The method of connecting to a Long Term Secret:

```
fromCalypso (Roster roster, SkipblockId byzcoinId, LTSId ltsId)
```

has been modified. This method connects to ByzCoin and then recovers the Long Term Secret. But with the previous optimization, the connection to ByzCoin is maintained. Therefore, a variant of this method was added to the library to take an instance of ByzCoin as input:

```
fromCalypso (ByzCoinRPC bc, LTSId ltsId)
```

This optimization avoids unnecessary reconnection to ByzCoin.

4.4 Memory profiling

After implementing the previous optimizations, the system performance was still very bad. It took me a while, but I finally figured out where the problem came from with memory profiling. It turns out that there is a problem with the management of objects created by cryptographic functions. To work, these functions use objects called GFp2, GFp6, GFp12 and TwistPoint. This four objects use BigInteger and BigInt to stored values. The problem is that these objects are instantiated but never deleted. As a result, the memory space allocated to them is never released and the application quickly run

out of memory as you can see in figure 2. To overcome the lack of memory, the garbage collector is launched and releases all objects that are no longer referenced. Since the application as only access to a few tens of megabit to store temporary objects and that so many big objects are created because of the complexity of the cryptographic functions, the garbage collector is launched almost every second. This is a big work overload for the application and therefore all communications with Cothority are extremely slow.

To address this problem, the library has been modified to include a pool of GFp2, GFp6, GFp12. Instead of creating a new object, an object is taken from the pool. When the object is no longer needed, it is cleaned and put back into the pool. The Miller function and all the functions it calls have been changed to use this pool. The result of this change is very positive because it has reduced by about three the number of objects allocated in memory. But that's still not enough. As you can see in figure 3, there is still a lot of objects instantiated and so a lot of calls to the garbage collector. The problem is that there is still many places in the library who does not use the pool.

The pool is a good solution but it has some drawbacks. First, the lifetime of each object has to be tracked down. Because of the complexity of cryptographic functions, it is not an easy task. In addition, the code becomes a little more complex because it must be explicitly specified when an object must be released.

5 Limitations

5.1 Link between user and signer

In the current implementation, one user can communicate with another because the public key and the value of his interlocutor are hardcoded in the user. It would be very convenient to have a mechanism to retrieve all the useful information of a user directly from Cothority.

For example, a user A could transmit his public key to a user B (for example via text message). User B could then use this key to ask Cothority to send him all the information related to the user A. Once user B has these data, he can easily communicate with him. Of course, it must first be defined what information can be transmitted to the user B in order not to create vulnerability in the system.

5.2 Parcelization limitation

Parcelization can only be used on android. Therefore, if someone wants to create a web interface to use the system from a computer, there will be a compatibility problem.

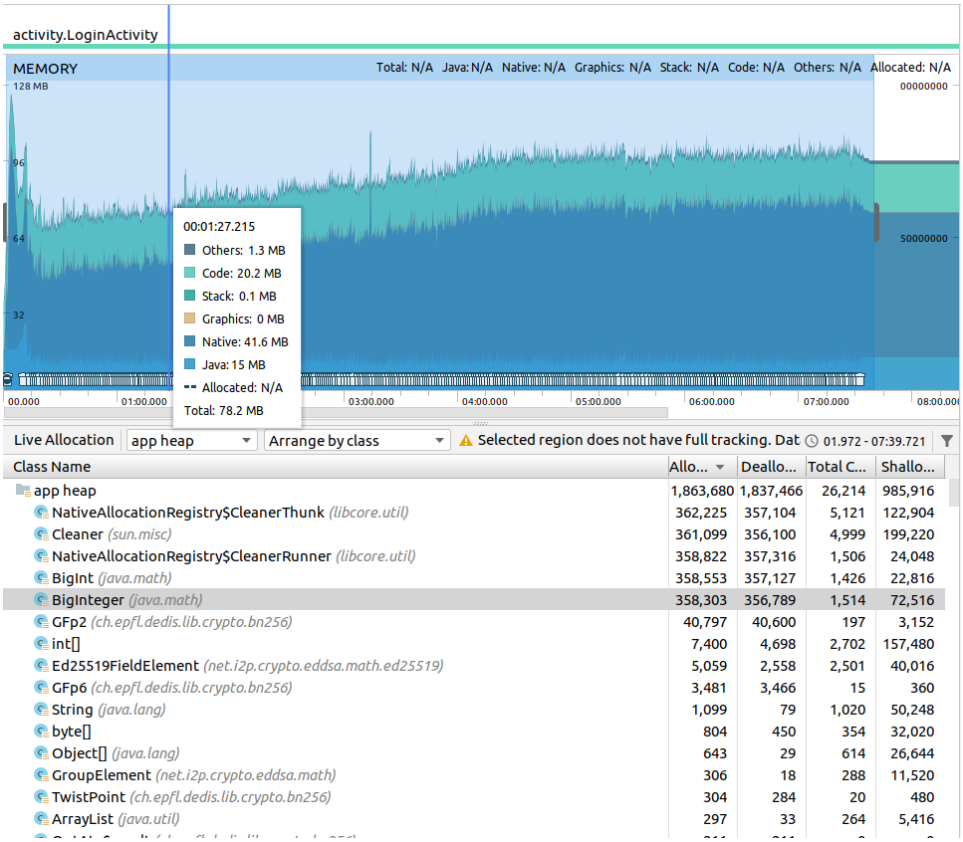

Figure 2: Memory profiling without pool

5.3 System performance

Since the Java library suffers from garbage problem, it is not possible to make performance tests. The results obtained would not reflect reality. However, the decentraliation with Cothority is certainly slightly less efficient in terms of speed than if it had been done with a classic server such as firebase. Indeed, before retrieving a document, it is necessary to communicate several times with Cothority: connecting to ByzCoin, sending and retrieving proofs, fetching data of the value instance, etc. While with a classic server a single roundtrip is needed to retrieve a document. However, a classic server does not have a such high level of security! And in addition, there is a way to reduce the number of communication with Cothority by storing locally the data. It is explained how to do this in the future work section.

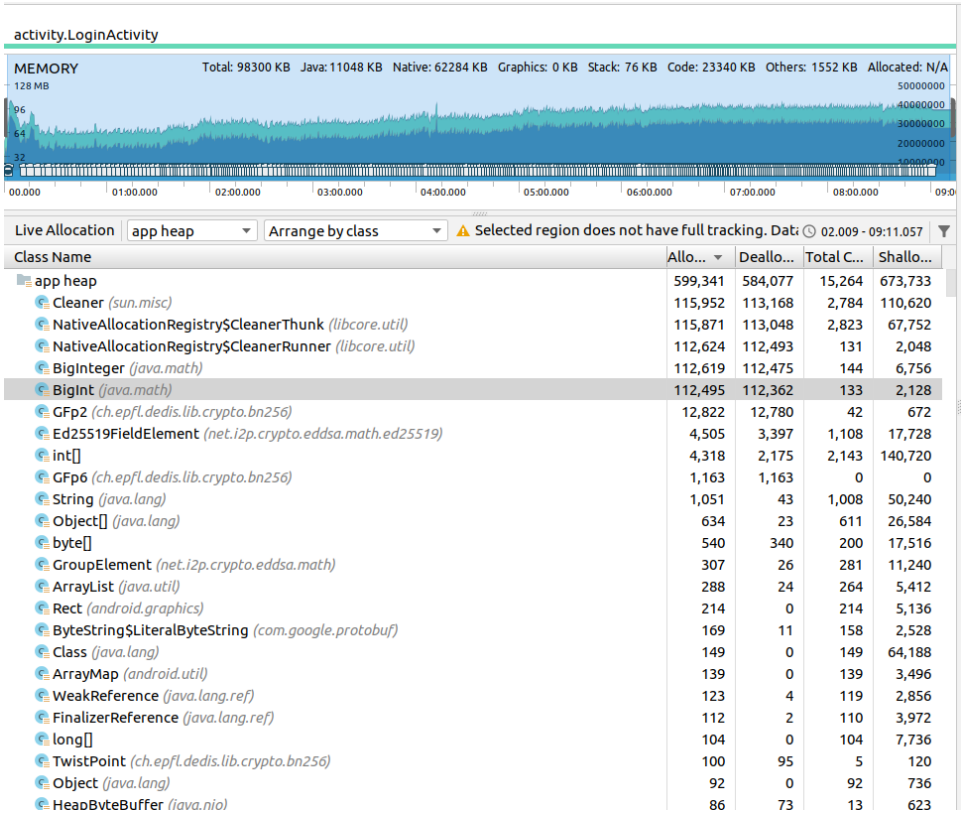

Figure 3: Memory profiling without pool

6 Future work

Since it took me some time to understand that the problem of slowness came from the garbage generated by the cryptographic functions, some of the initial goals have not been met: the modification and deletion of a document. That's why I'm going to explain here how to implement these two elements as well as other elements that may be interesting.

6.1 Modification of a existing document

Editing a document is almost identical to creating a new document. Once a document is on ByzCoin, it is no longer possible to edit it. It is therefore mandatory to create a new write instance of the document that has been modified. Then, the new id of the write instance has to be send to the user and the user must be warned that this is not a new document. For example add a bit to the message that told whether it is a modification or not. Finally, the user will spawn a new read instance and replace the old document ids by the new ones in memory. The old darc can be reused

without any modification.

6.2 Deletion of a document

Deletion is a bit more complicated to implement and need the library and the implementation to be modify a little bit. The publisher Darc must include a new rule allowing the user to spawn a Long Term Secret. Each time the user create a new document, he has to create a new LTS instance and to use it to publish the document on ByzCoin.

The publisher Darc must also contain a rule to delete an LTS instance. When the user wishes to delete the document, the deletion rule is invoked and sends a transaction that deletes the LTS instance. Once this is done, a proof is sent to the servers. They will use that proof to know that they are authorised to forget the secret key shard they are holding. After that the document is no longer accessible.

6.3 Local caching

Decentralization is a good thing but it has a cost. At each new connection to the application, the data must be reloaded. Therefore, a lot of mobile data can be used. If a user does not have an unlimited subscription, he can quickly exhaust his phone plan. In addition, if there are many users, the load on the server can be very large.

To overcome this problem, it is possible to store the data on the phone. To ensure security, the data is stored encrypted, and there is still a redundancy on the servers in case of problems.

The implementation can be done quite easily. First, a database where the encrypted documents will be stored has to be created. Then, the key material has to be saved in the android keystore. The keystore makes it possible to keep cryptographic keys securely. Once a key is in the keystore, it is not possible to recover it. However, it is possible to give encrypted data to the keystore and it will decrypt them. It is also possible to delete the keystore key. Moreover, it is even possible to restrict when and how keys can be used. For example, the user has to provide an authentication before been allowed to use the keystore.

When the user launch the application, he no longer has to download everything. The application must check that his data are up to date, decrypts them and check if new document has been shared with him.

To have more security, only the encrypted documents can be stored in the local cache. When the data need to be decrypted, the application must ask Cothority to send the key material.

6.4 Two more possible improvements

Currently, a unique key is used to encrypt all the documents of a user. It would be better to have one key per document and store the keys in the keystore.

A better mechanism of communication between the user should be implemented. For example, send the ids by text message. This would solve all the problems previously explained with the values instances.

7 Installation guide

A small Cothority running ByzCoin has to be created. The application does not have any special prerequisites to work and the dependencies are already present in the gradle file. The only configuration to do is to set up the the ip address of the cothority server in the cothorityUtils class.

To profile the memory, I used the memory profiler of android studio. Note that the profiler slows down the application considerably.

References

[1] github.com/dedis/cothority

[2] developer.android.com

[3] https://codedump.io/share/oTQ7FveHcufD/1/how-to-marshall-and-unmarshall-a- parcelable-to-a-byte-array-with-help-of-parcel

[4] http://androidopentutorials.com/android-sharedpreferences-tutorial-and-example/

[5] https://androidride.com/asynctask-android-tutorial-example/