ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Cross Platform Mobile Frontend for Blockchain Data Handling

Léo Meynent

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Bachelor Project

Autumn 2018

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Linus Gasser
EPFL / DEDIS

Contents

1 Introduction **1**

2 Background **2**
2.1 Cothority framework . . . 2
2.1.1 ByzCoin . . . 2
2.1.2 DARC . . . 3
2.2 BC Admin CLI . . . 4
2.3 PopCoins . . . 5
2.3.1 NativeScript . . . 5
2.3.2 Application UI . . . 6
2.3.3 Code structure . . . 6

3 Methodology **7**
3.1 BC Admin CLI . . . 7
3.1.1 Testing . . . 7
3.1.2 Documentation . . . 8
3.2 PopCoins . . . 8
3.2.1 Testing . . . 8
3.2.2 Documentation . . . 8

4 BC Admin CLI **9**
4.1 Spawning a DARC . . . 9
4.1.1 Context . . . 9
4.1.2 Implementation . . . 9
4.1.3 Avoiding key conflicts . . . 9
4.2 Getting a DARC from ByzCoin . . . 10
4.3 Managing DARC rules . . . 10
4.3.1 Context . . . 10
4.3.2 Implementation . . . 11
4.4 Sharing a ByzCoin config . . . 11
4.4.1 Context . . . 11
4.4.2 Implementation . . . 12
4.4.3 Limitations . . . 12

5 PopCoins **13**
5.1 User Interface . . . 13
5.1.1 ByzCoin main page . . . 14
5.1.2 Adding a new DARC . . . 14
5.1.3 DARC description page . . . 15
5.2 Local saves . . . 16
5.3 Scanning objects . . . 17
5.4 Creating client . . . 17

6 Limitations **18**
6.1 DARC rights management 18
6.2 Rapid deprecation of *PopCoins* code 19

7 Further work **20**
7.1 Rework ByzCoin config sharing methods 20
7.2 Implementing transaction management in PopCoins 21
7.3 Multi-signatures 21

A Appendix: Installation instructions **23**
A.1 BC Admin CLI 23
A.2 PopCoins 23

1 Introduction

This reports describes the work that has been done, for one part on the BC Admin tool, and, for the other part, on the cross-platform mobile application PopCoins. Both are interfaces allowing a user to interact with *ByzCoin* skipchains, features implemented as a part of DEDIS' *Cothority* framework. The project focused especially on the management of Distributed Access Right Controls (DARC): its main goal was to provide a terminal CLI[1] and a user-friendly mobile application to spawn, evolve and delete them.

To better understand the stakes and outcomes of this project, the report will begin with a short introduction on the *Cothority* framework and the *PopCoins* app.

[1]Command Line Interface

2 Background

2.1 Cothority framework

The apparition of BlockChain technologies has allowed easy and scalable collaboration between users and decentralisation of information. In that scope, *Cothority* allows to form *Collective Authorities* that run decentralised and distributed protocols on a set of *Cothority Servers*, a.k.a *Conodes*[2].

Cothority is implemented using a number of protocols designed to overcome the basic challenges of running algorithms in a decentralised way. They implement most of the consensus and cryptographic functions needed by the framework to properly work.

DEDIS has also implemented a number of applications on this framework, notably *Proof of Personhood*[3] and *ByzCoin*[4]. This project concerning mostly the second one, this report will not introduce *POP*.

2.1.1 ByzCoin

ByzCoin is an implementation of the *OmniLedger* protocol[5] designed by DEDIS lab at EPFL. It is a coin based on a *SkipChain*[6] that is designed to work on a *Cothority*. Its operation is better described by this picture (click).

The creation of a *ByzCoin* instance was already implemented at the beginning of the project. Based on a roster (its *Collective Authority*), it is created with a *Genesis DARC* and an *Admin Identity*. One of the goals of this project was to be able to create and manage other *DARC* than the *Genesis* one, for users who are not necessarily the Admin.

In order for a client to be able to interact with a *ByzCoin* instance, it needs to know its configuration. The content of the *Config* object may differ from implementation to implementation, but in any case it must contain:

- The *ByzCoin* instance ID
- The *Roster*, which is the set of *Conodes* on which the *ByzCoin* is deployed

It may also contain:

[2]`https://github.com/dedis/cothority`
[3]`https://github.com/dedis/cothority/tree/master/pop`
[4]`https://github.com/dedis/cothority/tree/master/byzcoin`
[5]`https://eprint.iacr.org/2017/406.pdf`
[6]`https://github.com/dedis/cothority/tree/master/skipchain`

- The *Genesis DARC*
- The Admin identity (public key)

2.1.2 DARC

One of the most important components of *ByzCoin* is *Distributed Access Rights Control*, a.k.a *DARC*. These structures allow a decentralised management of the access rights in the different instances of *ByzCoin*.

Each *DARC* holds the following information:

- Description (String)
- Version (Integer, the number of time its been evolved. It is sometimes also called Depth)
- ID (a SHA-256[7] digest of its own content)
- BaseID (the ID of its first version)
- PrevID (the ID of the *DARC* from which it was spawned)
- Rules (explained hereafter)
- Signatures (a set of signatures)

The rules are a set of action-expression pairs that regulate what can be done from a given *DARC*. While the action describes the instruction that can be signed, the expression describes using logical expressions which *DARC* or users must/can sign in order for the transaction to be valid.

Most used actions for *DARC* management include but are not limited to:

- *spawn:darc*
- *invoke:evolve*
- *_sign*

Expressions use ED25519[8] keys to identify the users. Using logical expressions allows them to require multiple signatures, or one signature amongst a set.

[7] `https://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html`
[8] `https://ed25519.cr.yp.to/`

In order to add a *DARC* to a *ByzCoin* instance, one must *spawn* it. A user who wants to spawn a *DARC* B first needs to have a *DARC* A on which he has the right to perform the action *spawn:darc*: it will be expressed in this report as spawning B *from* A.

DARC can have what is called an *owner*. The owner is a user having an exclusive access right on actions *_sign* and *invoke:evolve*. As such, he is free to sign with his *DARC* and evolve it to manage it as he sees fit. At spawn, *DARC* are usually created with an owner and comprise only those two rules, both linked with an expression composed only of the public key of the owner.

As *DARC* are part of the *SkipChain* blocks, they are immutable objetcs. In order to edit their settings, any user having the right to do so (through the *invoke:evolve* action) can *evolve* a *DARC*.

To evolve *DARC* A, the client first creates a copy - B. It then makes the changes the user desires on B, and increments the version number. As the content of the *DARC* will have changed, B will have a different ID from A (as it is a SHA-256 digest of the information it holds). However, A and B will still keep the same *BaseID*, which is the ID of their first ancestor, and will be identified by it. After having made these changes, the client puts B inside an *Instruction*, which it sends to *ByzCoin* through a *Transaction*. The *Cothority* then *spawns* B and will return it when asked for the *BaseID* of both A and B.

2.2 BC Admin CLI

The BC Admin CLI[9] is implemented in Go using the CLI V1 package from *gopkg.in*[10]. Some basic functionalities were already implemented at the start of the project:

- Creating new *ByzCoin* based on a given roster
- Evolving the *Genesis DARC* to add a rule
- Showing the *Genesis DARC* in the terminal

But these are not enough to correctly manage a *ByzCoin* and its access rights. The first part of the project thus was implementing some new functionalities to it, as will be described in section 4.

[9]`https://github.com/dedis/cothority/tree/master/byzcoin/bcadmin`
[10]`https://github.com/urfave/cli/tree/v1.20.0`

2.3 PopCoins

PopCoins is a *NativeScript* cross-platform mobile application developed by DEDIS lab[11] in order to propose *Cothority* applications in a simple, user-friendly way.

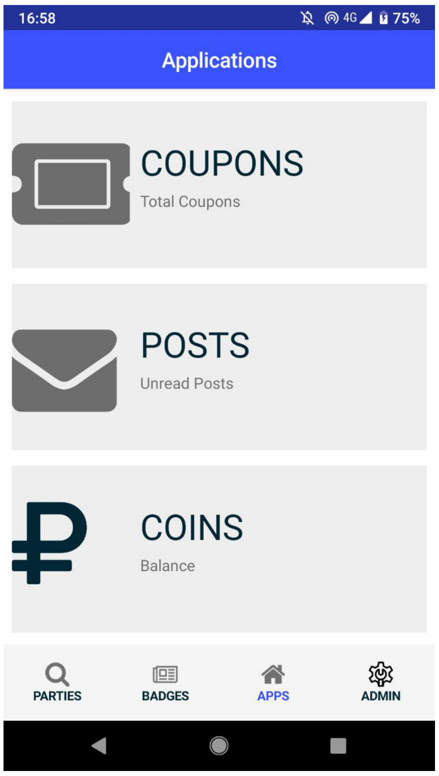


Figure 1: *PopCoins* main menu

PopCoins, at the beginning of the project, implemented features mainly linked with the *POP* application. As such, the features implemented during the course of this project did not modify the already implemented functionalities, as they were only remotely linked. It however reuses a lot of tools, such as those for the management of the *Conodes*, or the different clients and sockets linked with the *Cothority* services.

2.3.1 NativeScript

NativeScript[12] is a framework that allows to develop native cross-platform applications, compatible with both iOS and Android. It uses *JavaScript* for

[11]`https://github.com/dedis/popcoins`
[12]`https://www.nativescript.org/`

application logic, and a XML-CSS combination for the User Interface (UI). It also offers the opportunity to use native iOS and Android APIs through different bindings: this possibility has however not had to be used during this project

2.3.2 Application UI

The UI of the application is built around a first page describing the apps currently available. This is the most user-friendly area, where a non-initiated person should feel at ease. As such, details about *DARC* being quite technical, they have not been implemented in this part of the application.

An other part of the application is the Admin menu. It contains settings about coupons, new parties (both used for *POP*), and *conodes*. During this project, a *ByzCoin* part has also been added. More details about the UI as it was before the start of the project can be found inside the report of the student who designed it[13].

2.3.3 Code structure

The UI pages are coded inside the *pages* folder, which is itself organised around the structure of the UI. For this project, as only the admin interface has been modified, all UI changes can be found in *pages/admin/admin-page* and *pages/admin/byzcoin*. All of the pages are organised through modules, which are comprised of a *JavaScript* (eventually *TypeScript*) file for the logic part, and a *XML* file for the page description. It may also include a *CSS/SCSS* file for styling. Each files of the same module bear the same name with a different expansion.

The logic part of the application is situated inside the *lib* folder, which is itself divided by functionality. The files directly inside this root folder are the ones used at application-level. As many of the implemented changes are taking place at user-level, a big part of the logic implemented during this project has been appended directly to *lib/User.js*. Many of the more specific functionalities were used, notably *lib/network* and *lib/crypto*, but the most impacted one has certainly been *lib/cothority/omniledger*, in which changes have had to be made in order for it to be compatible with the latest versions of *Cothority* and *ByzCoin*.

[13]`https://github.com/dedis/student_18_xplatform/blob/master/report/report.pdf`

3 Methodology

3.1 BC Admin CLI

The whole code is to be found in the *bcadmin* folder of *cothority/byzcoin*[14]. Most of the changes have been made inside *main.go*, but a few were also made inside *lib/config.go*. Some other parts of *Cothority* have also been lightly changed, like for example *darc/darc.go*.

3.1.1 Testing

The choice has been made not to use *Go* unit tests for the testing of the code. Instead, we use a bash script implementing the *Cothority* library *libtest.sh* that interacts directly with the CLI. All of those tests are written inside the file *test.sh*.

DGB_TEST constant describes what outputs of tests will be shown: 0=none, 1=test-names, 2=all. By default keep at 1

DBG_SERVER constant describes the outputs written in terminal from server responses. By default keep at 0, but in case of an error on server-side it is always useful to put it at 2 for debug purposes.

NBR_SERVERS and NBR_SERVERS_GROUP are parameters for the tests, to be kept as-is.

testCreateStoreRead is a test that was implemented before this project. Most of the methods it checks are now deprecated.

testQR is the only non-automated test. The tester has to manually check the QR Code that is shown by *BC Admin CLI*

All other tests are fully automated and test the different commands added during this project

All those tests have been run and passed at the moment of the Pull Request.

[14]`https://github.com/dedis/cothority/tree/master/byzcoin/bcadmin`

3.1.2 Documentation

Detailed documentation is provided inside the *readme.md*[15] file. It describes all available commands as well as all of their optional flags. Also, documentation has been written inside the code for all new exported methods.

3.2 PopCoins

3.2.1 Testing

No unit test has been implemented for the code that has been written in the context of this project on *PopCoins*. However, each new feature has been thoroughly manually tested on Android. The tests have been realised on a physical device running Android 9.

It is to be noted that the new implementations have not been tested on either *Android Virtual Device* or iOS. Indeed, the current version having been judged stable enough, and specific equipment being required for iOS testing, the decision has been taken not to run the tests for both Android and iOS, which has lead Android to be the only tested environment.

In order to test network-related functionalities, the app has been tested with a test instance of *ByzCoin* created on a computer, connected to the physical Android device through USB tethering. This configuration allows communication between the two devices on a local network. All tests have been realised manually to check the correct operation of the application.

In order to create good testing conditions, a script[16] has been realised in order to run a local *ByzCoin* instance. It first creates the local *Conodes*, creates a *ByzCoin* instance and finally generates a QR Code representing its configuration.

3.2.2 Documentation

No specific documentation file has been written about the changes made on *PopCoins*. However, all of the methods used in logic part of the code have been commented directly inside the code. The code used for the UI has been judged verbose enough, as it is not supposed to be used elsewhere, and no specific comment has been added to it.

[15]`https://github.com/dedis/cothority/blob/master/byzcoin/bcadmin/README.md`
[16]`https://github.com/dedis/cothority/tree/master/conode/test_popcoins.sh`

4 BC Admin CLI

4.1 Spawning a DARC

4.1.1 Context

Before this project, BC Admin CLI could only spawn *DARC* that were evolutions of the *Genesis DARC*. The *darc add* command provides new options useful for *ByzCoin* access rights management:

- Spawning a *DARC* from any *DARC* the user chooses, not only the *Genesis* one
- Choosing the owner of the *DARC*
- Signing the transaction using any saved identity, not only the Admin one
- Outputing information about the newly spawned *DARC* in files in order to keep information and run automated tests

4.1.2 Implementation

The implementation of these new functionalities do not differ very much from the basic *add* command. The program retrieves the signer's keypair from local memory, the *DARC* from which we spawn a new one is retrieved from *ByzCoin*. Any error during those two steps stops the execution of the command and prints an error message. The public key of the owner is not checked, as it may have been generated from a different device. All these information are used to generate a transaction which is sent to *ByzCoin*.

The program does not directly check that that the given signer can use the given *DARC* to spawn a new *DARC*, as this check is performed directly by the *ByzCoin* service, which will return an error message if the transaction gets refused.

4.1.3 Avoiding key conflicts

In its current state, as described in sub-section 2.1.2, BC Admin CLI only allows spawning a *DARC* that will only have an *owner*, and no other rules. This raises the issue that two *DARC* created using the same public key as *owner* and from the same *DARC* (often the *Genesis* one) will have the same *BaseID* (as it is a digest of the content of the *DARC*), which they will keep even if they are evolved. However, *ByzCoin* cannot support multiple *DARC* sharing a *BaseID* if they are not evolutions of one another: it will thus reject the spawn instruction and send a "*key conflict*" error message.

To avoid such a case happening, the *DARC* are spawned with a random 32-bits integer in their description. The description being taken into account for the ID digestion, this solution allows to create multiple *DARC* with the same owner and from the same *DARC* while assuring a very small probability of key conflict.

4.2 Getting a DARC from ByzCoin

Before this project, BC Admin CLI only was able to manipulate the *Genesis DARC*, which was saved locally. In order to implement *DARC* management in the program, it became necessary to be able to get other *DARC*. As they have to be accessible to all users of *ByzCoin* and modifiable by them, it is not possible to only keep a save locally. Instead, one has to systematically fetch *DARC* information from *ByzCoin* service.

DARC are retrieved from *ByzCoin* through a *proof*. The client asks *ByzCoin* with the *BaseID* of the *DARC*, provided by the user. *ByzCoin* responds with a *proof* containing all information about the *DARC*, if it exists. This information is a *byte[] protobuf*, that can then be translated into a Go object.

This addition has allowed the implementation of the simple yet useful *darc show* command which prints or saves in a file all information about the *DARC* the user has demanded.

4.3 Managing DARC rules

4.3.1 Context

The previous implementation of BC Admin CLI did only allow the addition of rules to the *Genesis DARC*. This functionality is however not sufficient to efficiently manage *DARC*. Moreover, as *DARC* are created with only the rules *_sign* and *invoke:evolve*, it is necessary to be able to manage rules in order to be able to grant them any other right.

There are three possible operations on *DARC* rules: addition, edition and deletion. All of those are performed using the *darc rule* command with different flags:

- No flag: simple addition. If the action already exists, the command will fail
- -replace: if the action already exists, overwrites the expression with the one provided by the user. If it does not exist, the command will fail

- -delete: if the action exists, removes it from the *Rules Map()*. This flag has priority over the other command options, which means that if this flag is provided, all other parameters will be ignored and the rule in question will be deleted

4.3.2 Implementation

Addition and edition are extremely similar: their sole difference being that addition will prevent override and edition will not be able to act on a rule that does not exist yet. Those two actions are thus performed almost the same way, with only those checks and the *Map()* method used differing, and branching being determined by the presence or absence of the -replace flag.

Deletion is a little more different, as it is supposed to remove the rule altogether. It thus requires less parameters, as for example the expression. For this reason, it is coded inside a separate function.

In these three cases one part remains the same: the program will perform an evolution of the *DARC* whose rules are being changed. For this, as described in sub-section 2.1.2, it creates a copy of the *DARC*, applies the change of rules to it, increments the version number, and then sends it to *ByzCoin* to *spawn* it. In order for it to be possible, it is absolutely necessary for the user to sign the transaction with a keypair having the *invoke:evolve* right on the *DARC* whose rules are being changed; otherwise, the transaction will be rejected and the program will throw an error.

4.4 Sharing a ByzCoin config

4.4.1 Context

The previous implementation of BC Admin CLI allowed to create a *ByzCoin* and generate its config. It was however impossible to share this config with another user, or another device, or at least very uneasy. In order for a *ByzCoin* instance to be accessible by multiple users on diverse devices, it was necessary to implement a way to share the required information.

Apart from this first aspect, there is also the necessity for the management of *ByzCoin* instances to be able to log as the admin user on *PopCoins*. It means that besides the config, it is necessary to be able to share the admin keypair to another device.

In order to solve these issues, the solution that has been privileged is the usage of a QR Code. This method has been chosen in particular because the *PopCoins* app already implements a QR Code scanner, and because it is a very user-friendly solution, even for an inexperienced user.

4.4.2 Implementation

There are two possible cases when a user requires a QR Code: he wants the config, or he wants the config together with the admin keypair in order to be logged in as the admin user. In the *qr* command of BC Admin CLI, these cases are differentiated by the presence or absence of the flag -admin. If present, the QR Code will also contain the admin keypair.

In order to generate the QR Code, the program uses the library *qrgo* by *qantik*[17], which is already in use in *CISC Identity Skipchain* (another *Cothority* application developped by DEDIS). This library allows the creation of QR Codes representing a string, and its display in the terminal. This allows an user wanting to save his QR Code to simply use the command *bcadmin qr > myfile*", and then "*cat myfile*" to print it again. (on Unix systems)

The string encoded into a QR Code is a JSON representation of a struct that contains only the necessary information. If -admin is not specified, it only contains the *ByzCoin* ID. If it is specified, it also contains the admin keypair. The device that scans the QR Code will then only have to read the QR Code in order to retrieve the string and parse it.

4.4.3 Limitations

As the QR Code can only hold a limited amount of information, it cannot contain the description of the roster. As such, at the moment, the device that will be scanning the QR Code should already have been configured to use the right roster of *Conodes*. This is a strong limitation as half of the configuration of the *ByzCoin* instance still has to be manually performed by the user.

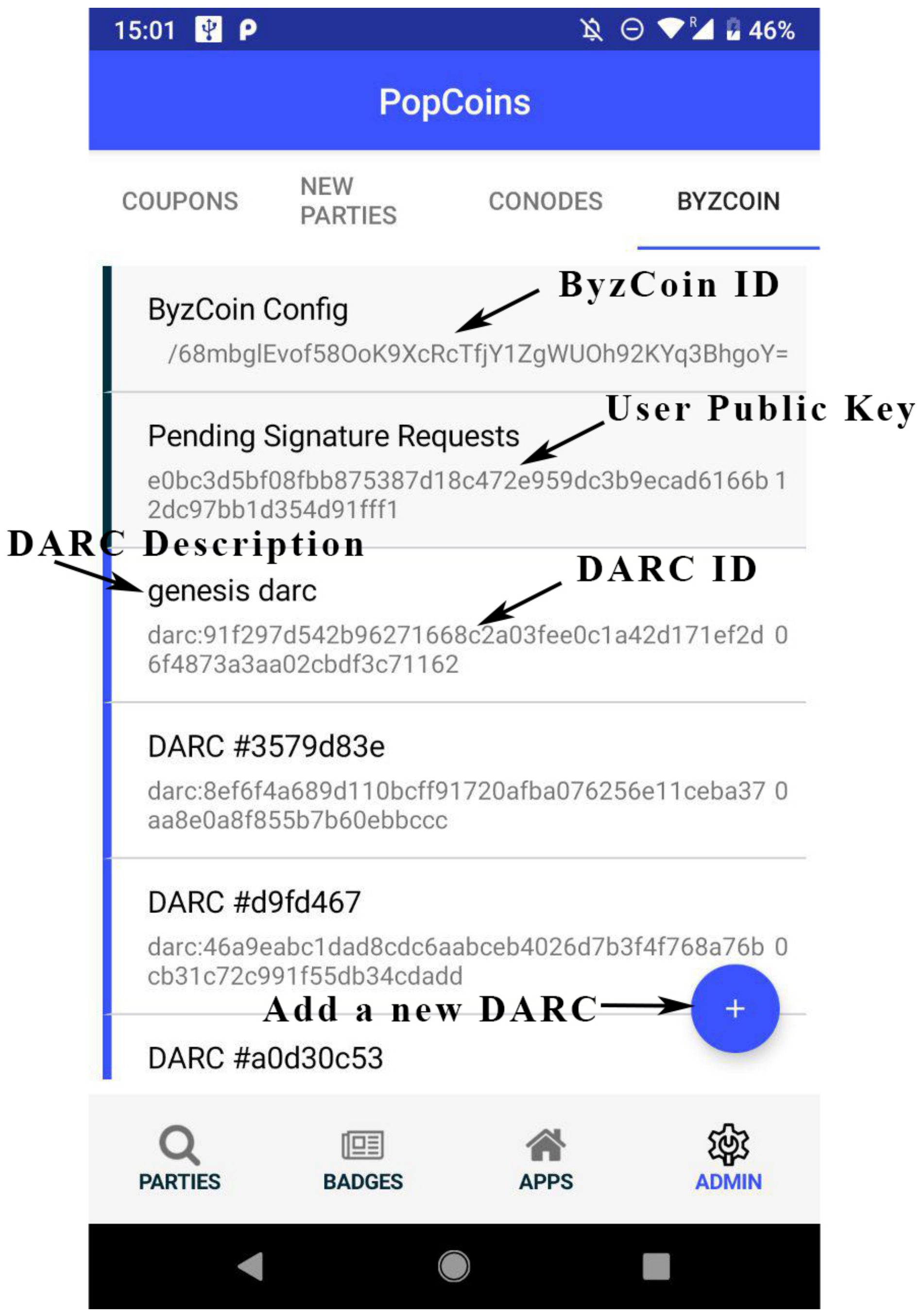
Also, in the current state of the BC Admin CLI, we transfer the admin keypair (which includes the private key) completely un-encoded through a simple QR Code. Anyone having access to the CLI where the admin is logged, or to any save or picture of the concerned QR code can take control of the *Genesis DARC*, which is particularly dangerous.

For those reasons, the current implementation of this functionality is to be considered as temporary and work in progress. It is useful for development purposes, but cannot be kept as-is for a release version of *ByzCoin*.

[17]`https://github.com/qantik/qrgo`

5 PopCoins

5.1 User Interface



ByzCoin Config
/68mbglEvof58OoK9XcRcTfjY1ZgWUOh92KYq3BhgoY=

Pending Signature Requests
e0bc3d5bf08fbb875387d18c472e959dc3b9ecad6166b12dc97bb1d354d91fff1

genesis darc
darc:91f297d542b96271668c2a03fee0c1a42d171ef2d06f4873a3aa02cbdf3c71162

DARC #3579d83e
darc:8ef6f4a689d110bcff91720afba076256e11ceba370aa8e0a8f855b7b60ebbccc

DARC #d9fd467
darc:46a9eabc1dad8cdc6aabceb4026d7b3f4f768a76b0cb31c72c991f55db34cdadd

DARC #a0d30c53

Figure 2: *ByzCoin* menu in *PopCoins*

5.1.1 ByzCoin main page

ByzCoin main page in *PopCoins*, situated inside the Admin panel, is built around two lists:

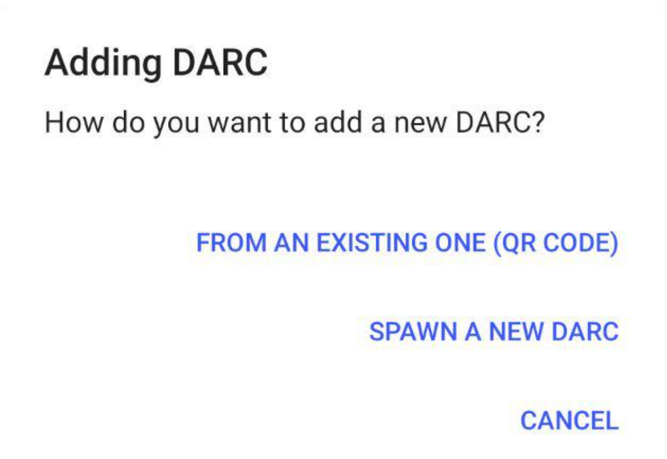
- The first list is of fixed size, and is composed of two elements which lead to sub-menus. The first element is the *ByzCoin* config, and leads to a description of it. The second element is only a stub in the current state of the project. Its a a menu meant to contain signature requests for the multi-signature implementation. This first list is differentiated from the second one through the use of a different colour on the side, declared in CSS as:
  ■ *night-dark*, RGB = 2, 49, 65
  The colour of the background is also a little bit darker:
  □ *background-dark*, RGB = 248, 248, 248

- The second list is dynamic, and shows the different *DARC* locally saved by the user. When the user taps a *DARC*, it leads to a page with its complete description. This list is identified by the colour:
  🟦 *accent-dark*, RGB = 58, 83, 255

There is also a floating blue button, with a white + cross, floating in the bottom-right corner of the page, and which is used to add a new *DARC*. The second list is fully scrollable in the event it exceeds screen size, which allows the elements a the first list to be always visible.

5.1.2 Adding a new DARC

Upon clicking on the white + button, a dialog pops-up in the app and presents the two different options available to the user:



Adding DARC

How do you want to add a new DARC?

FROM AN EXISTING ONE (QR CODE)

SPAWN A NEW DARC

CANCEL

Figure 3: *D*ialog offering to add a new *DARC* in *PopCoins*

- From an existing one, using a QR Code. This option will open up a QR Code scanner the user can use in order to read a QR Code generated from *PopCoins* app on another device. The whole scanning process is described in sub-section 5.3. If the user scans the QR Code of a *DARC* A that has the same *BaseID* as a *DARC* B stored locally, *DARC* A will replace *DARC* B if and only if its version number is strictly superior to B's; otherwise, A gets dropped.

- Spawning a new *DARC*. In that case, *PopCoins* will automatically generate the *DARC*, with the current user as its owner, and the string "*DARC #*" followed by a random 8-digits hexadecimal number as description (for the reasons explained in sub-section 4.1.3). In the background, the app will check that the user has at least one *DARC* on which he has the right to *spawn:darc*, otherwise the operation will fail. If the user has multiple *DARC* with the *spawn:darc* right, the first one in the list will be used for the transaction.

5.1.3 DARC description page

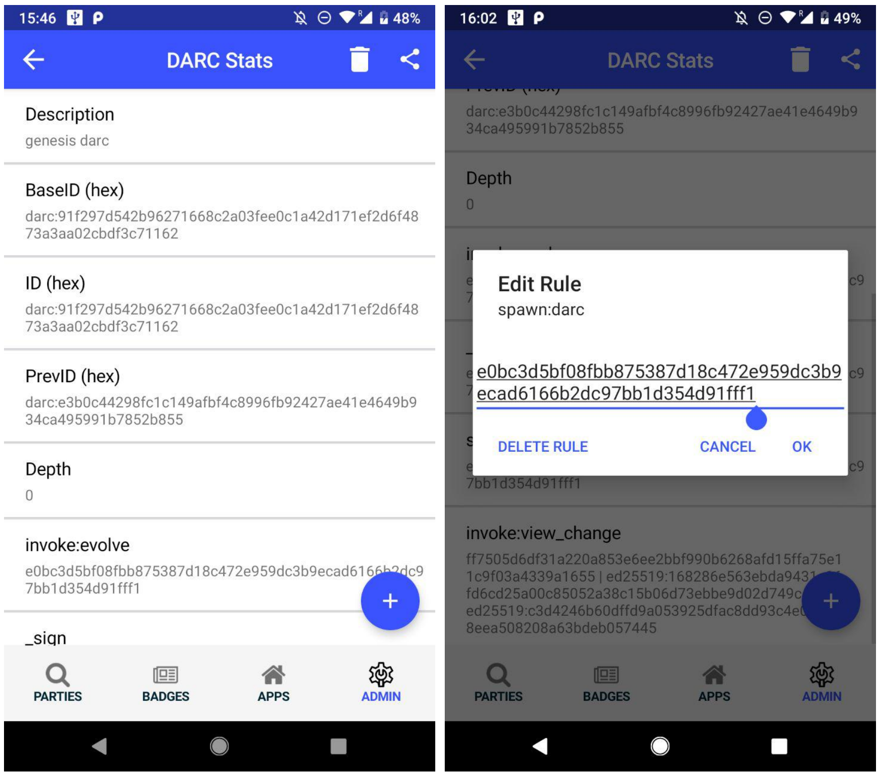
The *DARC* description page is composed of a single, scrollable list that shows all of its information. It ends is a dynamically generated list if its rules, identified by the action in the title and the linked expression in the content of the list element. On top, there is a button to go back to the *ByzCoin* main page, a button to delete the *DARC* from local memory (not from *ByzCoin*!), and a button to generate the QR Code representation of the *DARC*. In the bottom-right corner, there is a floating button containing a white + that allows the addition of new rules.

Whenever a rule is clicked, a dialog pops-up allowing the user to either delete the rule, or edit it. The rules *_sign* and *invoke:evolve* cannot be deleted. The description of the *DARC* can also be edited by the user the same way. In any case, for addition, edition or deletion, the user needs to have the *invoke:evolve* right on the *DARC*, as in the background, any change of that kind performs an evolution of the *DARC*. If the evolution process fails, an error dialog appears and the change to the *DARC* is cancelled.

The *ByzCoin* config description page is very similar to the *DARC* description page. There are only a few differences:

- If no config is saved locally, the delete and share button are replaced by a + button that will allow to scan a config QR Code

- There is no possibility to either add, edit or delete config parameters. The user can only completely delete the config to load a new one



Description
genesis darc

BaseID (hex)
darc:91f297d542b96271668c2a03fee0c1a42d171ef2d6f48
73a3aa02cbdf3c71162

ID (hex)
darc:91f297d542b96271668c2a03fee0c1a42d171ef2d6f48
73a3aa02cbdf3c71162

PrevID (hex)
darc:e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b9
34ca495991b7852b855

Depth
0

invoke:evolve
e0bc3d5bf08fbb875387d18c472e959dc3b9ecad6166b2dc9
7bb1d354d91fff1

_sign

darc:e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b9
34ca495991b7852b855

Depth
0

Edit Rule
spawn:darc

e0bc3d5bf08fbb875387d18c472e959dc3b9
ecad6166b2dc97bb1d354d91fff1

DELETE RULE CANCEL OK

7bb1d354d91fff1

invoke:view_change
ff7505d6df31a220a853e6ee2bbf990b6268afd15ffa75e1
1c9f03a4339a1655 | ed25519:168286e563ebda9431
fd6cd25a00c85052a38c15b06d73ebbe9d02d749c
ed25519:c3d4246b60dffd9a053925dfac8dd93c4e
8eea508208a63bdeb057445

Figure 4: DARC description page in *PopCoins*

5.2 Local saves

PopCoins has to save some information inside the device's memory in order to keep it between two runs. Those saves are made in the form of JSON[18] files, which locations are registered inside constants in the *FilePaths.js* file. During this project, two new elements had to be saved by the application: the *ByzCoin* config and the registered *DARC*. As for the *Conodes* and other user information, those saves are performed inside *User.js*.

One of the main issues caused by saving in JSON format is that *JavaScript* does not support well the stringification of *Map()* objects, which are used to represent *DARC* rules. In order to solve this issue, the solution that has been chosen is to create a temporary *DARC* object during the saving process, that instead of a *Map()* uses an *Array()* of tuples. This allows an easy stringification while keeping the *Map()* structure for *DARC* objects, which is extremely practical for the usual operations on rules.

[18]`https://www.json.org/`

5.3 Scanning objects

The main tool chosen for communication between devices connected to a same *ByzCoin* instance is QR Code. This solution is extremely practical to share objects between mobile devices, and from computers to mobile devices. *PopCoins* being the principal client aiming unexperienced users, this solution is particularly viable.

QR Code generation and scan were already implemented in the app before the start of this project, and used for example with the *conodes*. It is built around the *nativescript-barcodescanner* plugin for *NativeScript*[19]. During this project, QR Code generation has been implemented for two new objects: *ByzCoin* config and *DARC*. As such, it is really easy for a user B to join the same *ByzCoin* instance as user A: user A only has to generate QR Codes of his *Conodes*, his config and eventually some of his locally saved *DARC*.

The main challenge comes from the interaction via QR Code between BC Admin CLI and *PopCoins*: both generate their QR Codes from JSON representations of the objects, but they do not represent those objects the same way. Decision has thus been taken that for the *ByzCoin* config, the representation generated by BC Admin CLI would be the one used. It thus follows that in *PopCoins*, a special parse algorithm has been implemented in order to translate the Go representation of the config object into the JavaScript representation of this same object. Equivalently, *PopCoins* will make the inverse translation when generating the QR Code of the config, which will be represented the same way as in BC Admin CLI. As BC Admin CLI does not generate QR Codes for *DARC*, the default string representation of *DARC* is the one from JavaScript.

5.4 Creating client

In order to communicate with a distant *ByzCoin* instance, *PopCoins* needs to create a client object to handle the communication between the application and the distant service. In order to do that, the application first creates a socket with its roster of *Conodes*, and then attempts a handshake with the *ByzCoin* service. If this operation works, the client loads information from the service and then is ready for use.

During this project, only the part described above has been implemented. Due to deprecation of some JavaScript networking and transaction building modules, the application still is unable to send any transaction through this client. This part will be developed in sub-section 7.2.

[19]`https://github.com/EddyVerbruggen/nativescript-barcodescanner`

6 Limitations

6.1 DARC rights management

DARC are designed in a form that allows them to be fully personalised by their owners. But in their actual form, not only can they be personalised, but their owner, having the right to evolve them as they see fit, can grant themselves any right they want without restriction.

As an example, we could imagine an admin A who wants to keep the exclusivity of the right to spawn new *DARC* (e.g. *spawn:darc* rule). He then creates a new *DARC* for user U. User U being the *owner* of his own *DARC*, he has the right to evolve it (e.g. *invoke:evolve*). However, in the current state of *ByzCoin*, U is completely free to evolve his *DARC* to add the rule *spawn:darc* to it.

Thus, any user that owns a *DARC* virtually has all rights a *DARC* can grant. The only limitation is that it cannot grant access to any other *DARC*.

In order to avoid such a scenario, there are two possibilities:

- Not giving the owner of the DARC the *invoke:evolve* right, but instead giving it to the couple "admin & user". This way, the admin would have entire control on which rule he wants or not to be added to the *DARC*. The main downside would be that the admin would have to check every evolution made on the *DARC*, including for example minor changes to the description. Moreover, this would mean that the *Distributed Access Right Controls* would be entirely centralised around the admin of the *ByzCoin*, who would have to check every transaction, which would cause a big problem of scalability.
- Adapting the *DARC* contract to limit the ability of the *owner* to add any rule he wants. An avenue would be not allowing a *DARC* to grant more right that his previous one (the *DARC* pointed by the *PrevID*), which would also assure that any *DARC* the user spawns afterwards could not, in any case, have more rights than the first *DARC* the user has been granted. This would allow the admin to only manage the rights of the "parent" *DARC*, any *DARC* below it having the impossibility to have more rights than it while still being entirely manageable by their respective owners. However, this would allow a situation in which, all users having the right to *spawn:darc* (because the parent *DARC* has to have it), any user could create *DARC* for new users, making it impossible to control who can have access to a *DARC*. Moreover, this method would be conflictual if the admin decides to remove rights to the parent *DARC*: all *DARC* below

would have to also lose this right, but the admin does not necessarily have the right to evolve them, and it would imply to implement an efficient process to retrieve all *DARC* below the parent one.

6.2 Rapid deprecation of *PopCoins* code

PopCoins has been developped in order to be able to communicate with the *ByzCoin* service, implemented in Go inside the *Cothority* framework. The *Cothority* framework is itself based on multiples services coded by DEDIS lab, as for example *kyber*[20] for cryptographic operations, *onet*[21] for networking...

All of those services being in development, they tend to change very regularly, sometimes influencing relatively heavily the way the services interact. Because of this, *PopCoins*, as it has to interact with *ByzCoin* through the network, is strongly affected by those most significant changes. As it is coded in another language and uses adaptations of the Go code in JavaScript[22], it requires important efforts to be kept up-to-date with the most recent implementations of the *Cothority* framework.

Such an issue arose in the late stage of this project: important changes to *cothority* and *onet* rendered *PopCoins* implementation incompatible with the latest version of the framework. The project being in its last days when the error has been discovered, it has not been fixed, and the application needs to be tested on an older version of the framework in order to work.

[20] `https://github.com/dedis/kyber`

[21] `https://github.com/dedis/onet`

[22] `https://github.com/dedis/cothority/tree/master/external/js`

7 Further work

In this section, the future work to be done on this project is ranked from most to least critical.

7.1 Rework ByzCoin config sharing methods

As described in sub-section 4.4.3, the current solution for sharing *ByzCoin* config between devices is limited in functionality and unsecure. It is relatively critical to change it as it could compromise the admin keypair, and thus the *Genesis DARC*.

For the first issue, which is sharing roster information together with the config, an idea would be to only transmit the IP addresses of the conodes. This solution however presents the downside of being difficult to scale: if the roster is comprised of a few tens of nodes, it will quickly be too much IP addresses to put into a simple QR code. In that case, a mitigation would be to give the IP of one *conode* and then retrieve the whole roster from it.

The second issue is harder to solve: we want to share a private key across devices, which is pretty much against the principle of a private key. There are two possible mitigations:

- Encoding the private key for the transfer between the two devices. The main downside of this solution is that it is not 100% secure, and that it will be hard to ensure both confidentiality and authenticity. Guaranteeing those security properties would necessitate repeated communication between the two devices, which would be impractical with QR codes

- Creating a second keypair for the second device, and thus avoid transmitting the admin keypair. But here again, in order to be granted the rights to the *Genesis DARC* with the second identity, one would have to ensure authenticity, which is difficult with QR Codes for the abovementioned reasons. Moreover, it would imply that *ByzCoin* should always check that all references to one of those two identities is linked with the other with a logical "or", which seems also really difficult to implement and scale.

Those solutions being not really satisfying, the only remaining possibility would be to completely rework the process used to authenticate a same user on two different devices. In an ideal case, the user should enter manually his private key inside each of his devices; however, as *ByzCoin* uses ED25519 keys, it is impossible for a human being to simply remember it, and laborious to write it manually. Sharing an identity on multiple device would thus imply to use an additional authentication system in order to securely work.

7.2 Implementing transaction management in PopCoins

As described in sub-section 6.2, the current implementation of *PopCoins* is not up-to-date with the latest version of the *Cothority* framework, and will first need to be updated in order the client to communicate with the latest implementation of *ByzCoin*. Apart from that, as described in sub-section 5.4, *PopCoins* is able to create a client to interract with a *ByzCoin* instance, but still cannot send transactions through it. The cause of it is the deprecation of some JavaScript networking and transaction-building modules, that are not up-to-date with the latest changes made on the framework in Go. For now, *PopCoins* is thus completely unable to interact with the *ByzCoin* instance it is connected to.

In order to solve this issue, one will have to update the code for the latest standard of *ByzCoin* as it has been coded in Go, and will need to implement transactions in the relevant functions of the code.

Currently, transactions are only needed for two different kind of instructions: spawning a *DARC* and evolving a *DARC*. Functions relative to these functionalities are already implemented inside *User.js*, and code comments mark where transaction management should be added in the related functions, repectively *spawnDarc()* and *evolveDarc()*.

7.3 Multi-signatures

Expressions linked with rules, as explained in sub-section 2.1.2, can express sets of required signatures using logical *or* and logical *and*. It means that in some cases, it is possible to require multiple signatures for a transaction to be accepted. However, in the current state of the project, in both BC Admin CLI and *PopCoins*, it is impossible to sign a same transaction with multiple identities.

In BC Admin CLI, the commands do not provide any option to sign transactions multiple times. However, the program being able to handle multiple identities, it should not be too difficult to implement such a feature if all of the identities are stored locally. In *PopCoins* however, the application is designed to handle a single user, and as such a single keypair.

In both cases, it is impossible to sign a same transaction from different devices. If we consider the base case, which would be multiple users logged in on their respective devices that need a collective signature for a transaction, such scenario is unsolvable at the moment. It is thus necessary to implement inside *ByzCoin* service a protocol to share transactions between devices before sending them to *ByzCoin*. An idea could be for *ByzCoin* to keep pending transactions in memory, and users would regularly download transactions waiting for their signature and accept or refuse to sign them. When a transaction has enough signatures, it gets sent to *ByzCoin*; if this state becomes unreachable because of too much refusals or a timeout happens, the transaction gets dumped.

Another issue concerning this is that Expressions still are not implemented inside *PopCoins*. It would thus be necessary to implement a JavaScript program able to resolve expressions, which can be based on its Go counterpart[23].

[23]`https://github.com/dedis/cothority/tree/master/darc/expression`

A Appendix: Installation instructions

A.1 BC Admin CLI

BC Admin CLI requires Go: detailed installation instruction can be found on their official website (click).

Then, you will need to get the cothority framework from github:

```
$ go get github.com/dedis/cothority
```

Go should automatically install the dependencies. In order to check that BC Admin is working correctly, try to run *test.sh*.

You can easily update the executable *bcadmin* inside the bin folder of go by running the following commands inside the *bcadmin* folder:

```
$ go build -o bcadmin ./*.go
$ cp bcadmin \$(go env GOPATH)/bin/
```

A.2 PopCoins

First, you will need to install *NativeScript* framework: detailed installation instruction are available on their official website (click).

Then, clone the *PopCoins* repository from github.com:

```
$ git clone https://github.com/dedis/popcoins
```

Afterwards, go inside the *popcoins* folder and run:

```
$ make start-dev
```

Once this is done, you are ready to go! Plug-in your physical device or start your emulator and run one of the following:

```
$ tns run android
$ tns run ios
```

The application should launch on your device/emulator