DynaCred on ByzCoin

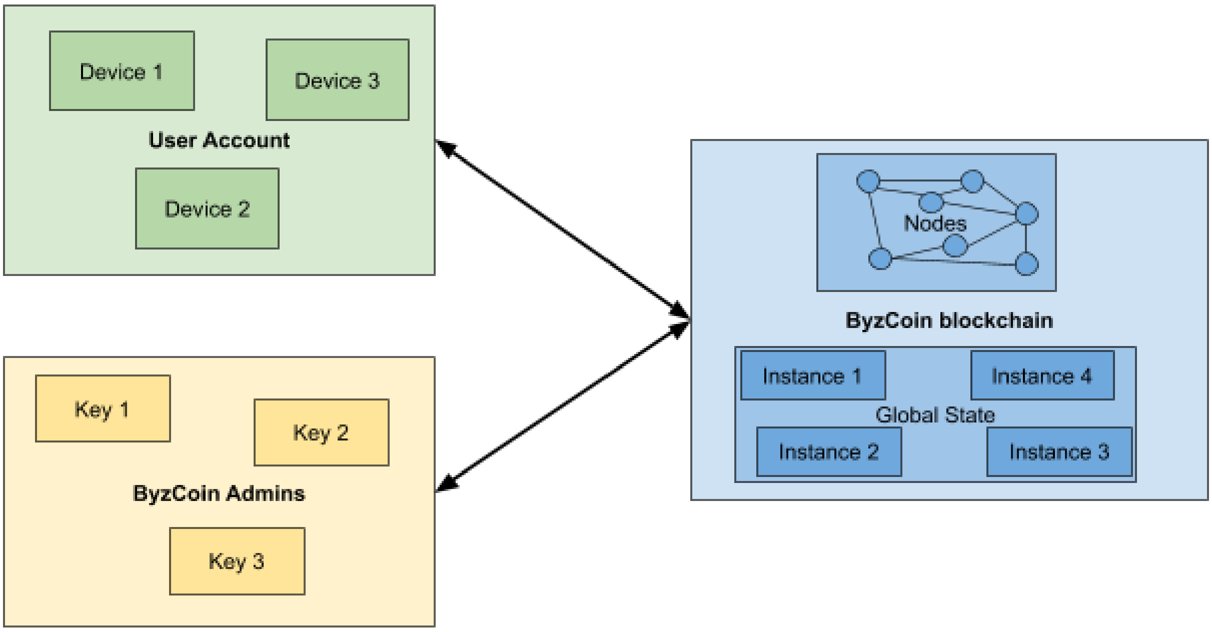
The ByzCoin network has a test-net up and running, and first applications are starting to use it. Currently there are a lot of open “work in progress”, but it’s running stable. One of the first applications that started on the DEDIS blockchain was the personhood app, providing a simple way of setting up personhood-parties. Following the personhood-parties came the effort of providing an identity service that includes multiple sources to create a reliable proof that a certain account is held by a human being instead of a bot.

In the C4DT, an effort has been started to present a *demonstrator* of the omniledger technology (which is currently only a byzcoin implementation), and the first use-case of this demonstrator is a decentralized credential system that is used for the login service to the restricted parts of the c4dt-website and the matrix-chat.

This document describes the different technical elements of this login system, and how it uses byzcoin contracts to implement users.

Parts of the system

The following figure shows a very rough overview of the different system parts that will be described in this document:

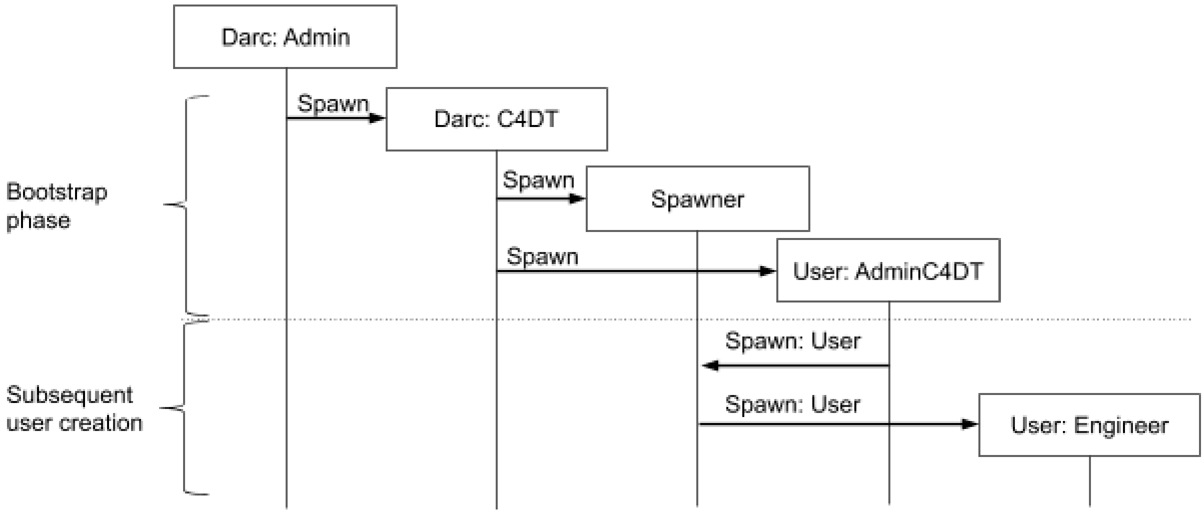


- ByzCoin blockchain: a set of nodes come to a consensus every n seconds on what the new *global state* of the chain should be. The global state consists of instances. Every instance:
    - is linked to a contract defining what actions can be done (with a few exceptions like signer counters that don’t have a contract attached)
    - has a darc attached to it that defines the rules to execute the available actions

    - has data that is stored in all nodes
- ByzCoin Admins are set up at the beginning of a byzcoin blockchain and can evolve over time. The admins rarely interact with the chain. If they do, they commit one of the following actions:
    - Add or remove a node to the system
    - Add or remove an admin to the system
        - Every admin can have different access rights
- User accounts are a set of different instances linked together as described in the rest of this document

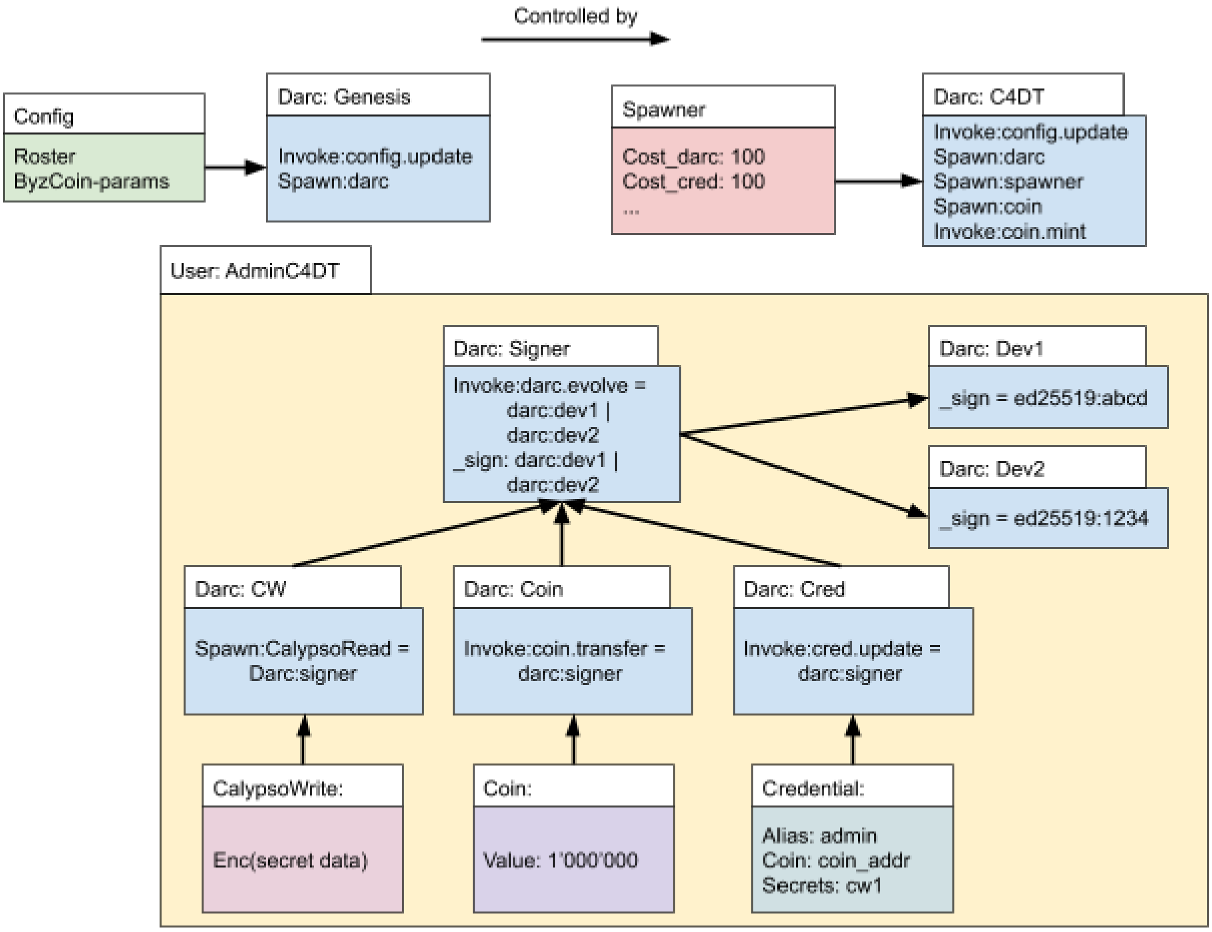
Life-cycle of the System

When a new ByzCoin system is set up, there exists an *Admin* darc, which is currently held by the DEDIS lab at EPFL. In a first *bootstrap* phase, the C4DT darc, the spawner, and the first user are created.
Every subsequent user creation uses the spawner instance to spawn all other needed instances.



Bootstrap Phase

Out of the Admin darc, a *C4DT* darc has been spawned and given to the C4DT. This darc is a universal darc that is allowed to spawn whatever instances it wants. The only difference with the Admin darc is that the C4DT darc cannot change the configuration of the ByzCoin blockchain.
In a first step, the C4DT darc is used to create a spawner instance and the AdminC4DT user. The spawner darc can be configured with the costs of the different instances it creates, see Appendix A. The first user is created directly by the C4DT darc and has exactly the same instances as every subsequent user.
The final picture within the *Global State* of ByzCoin after the bootstrap phase looks like this. Every box in this picture is an instance in the Global State of ByzCoin:

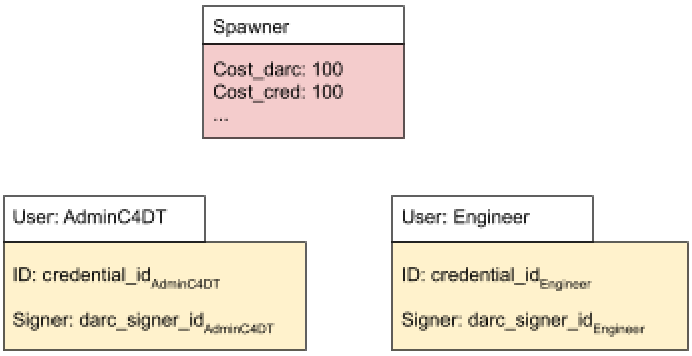


The following observations can be made about the user:

- The user already has 2 devices setup that are allowed to sign on behalf of the *signer* darc
- Each instance has its own darc that defines the rules specific to that instance. This is mostly for easier handling in the backend and to separate the devices-darcs from the other darcs.
- Each device darc represents one device that the user holds. A user can give access to his structure by adding a new device-darc with a new public key, and changing the *_sign* and *invoke:darc.evolve* rules of the signer darc
- The *signer* darc is referenced by all the other darcs in the user.

Subsequent User Creation

Once the bootstrap phase is over, the C4DT darc is not needed anymore for user creation, but only to mint coins for the users.
To create a new user, the AdminC4DT user can directly instruct the spawner instance to create all the necessary instances. The spawner has a list of costs for the different instances it can spawn. Once the new user has been created, part of the global state looks like this:


Spawner
Cost_darc: 100
Cost_cred: 100
...
User: AdminC4DT
ID: credential_id$_{AdminC4DT}$
Signer: darc_signer_id$_{AdminC4DT}$
User: Engineer
ID: credential_id$_{Engineer}$
Signer: darc_signer_id$_{Engineer}$

The current implementation uses only one type of coin for every cost, but the C4DT darc could decide that the different costs are covered by different types of coins, allowing to give only certain coins to users who then could only spawn certain instances.

User Creation Details

Following are some details with regard to the creation of a new user by the spawner.

Costs

For a standard user, the following cost occurs:

- 4 darcs = 4 * 100
- 1 coin = 1 * 100
- 1 credential = 1 * 1’000

For more details, look at Appendix A. This is somewhat arbitrary, but reflects the idea that a credential is more load to the system than a darc or a coin, as the credential will be updated more often and can grow larger in size.

Ephemeral Private Key

The standard setup of a new user should be like this:

1. The new user chooses a keypair and sends the public key to the admin
2. The admin creates the new user using the public key

However, to avoid the user having to send something to the admin, the current code does the following:

1. The admin
    a. choses an ephemeral keypair
    b. creates the new user with this keypair
    c. Sends the private key to the new user
2. The new user
    a. Choses a random keypair
    b. Evolves the device darc and replaces the ephemeral keypair of the admin with his own keypair

This allows the admin to directly send a signup link to a new user, without having to do a back-and-forth between the new user and the admin. Once the device darc has been updated to the new keypair, the admin cannot change the darc anymore.
If the system should allow recovery of the user by the admin, the user signer darc can point to a *recovery device darc* that holds the public key of the admin. If needed, the admin can then evolve the signer darc to change the devices of the user.

User Identities

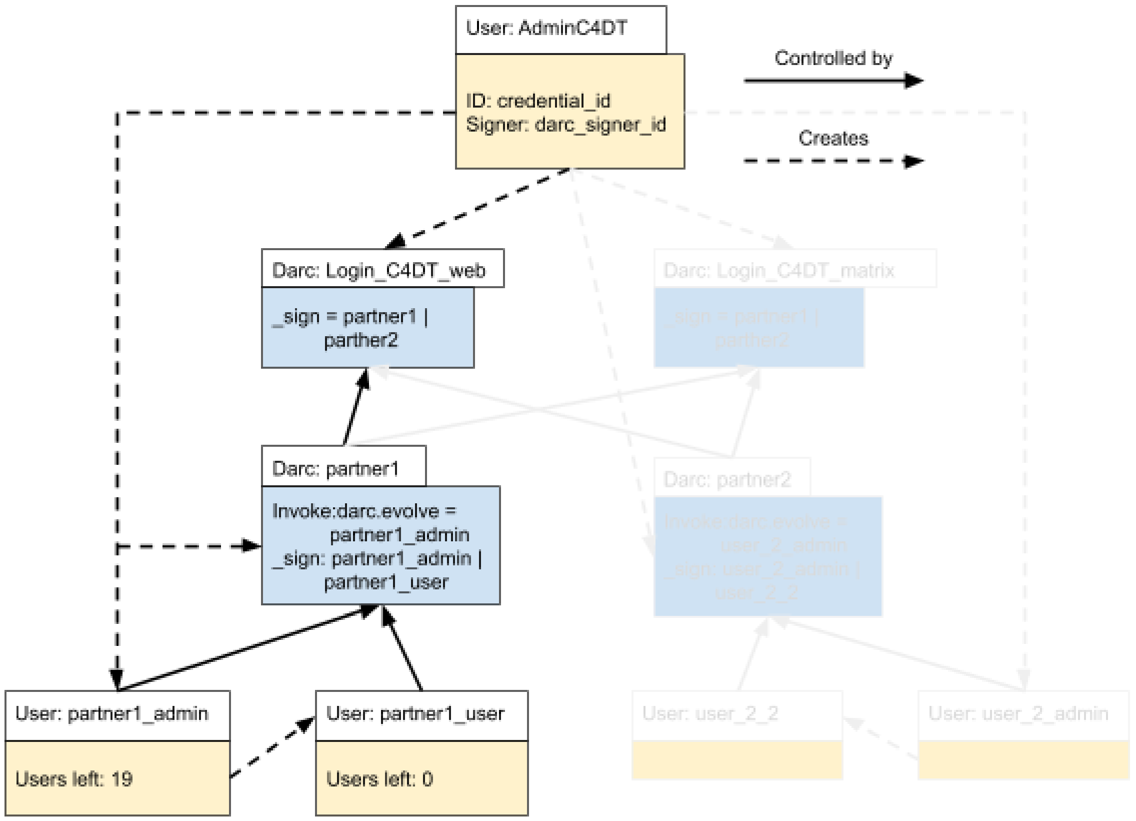
The global ID of a user should be the *credentialID*, which is the instance ID of the credential instance, because it allows to find all the different pieces of the user. This id should be used when exchanging contacts.
For referencing a user in a darc, the *darc:signer* should be used, because it represents all the devices the user has access to. If an application has the credentialID, it can get the credential-darc and the signer-darc very easily. The inverse is not always possible.

Creation of Actions and Groups

Darcs are very general and allow for multiple use-cases. For the C4DT demonstrator, we use them to implement the following two use-cases:

- *Action*, which is not tied directly to an instance per se, but describes in its *_sign* rule who is allowed to execute a certain action outside of the system. For the moment we use two actions in the C4DT demonstrator:
    a. *Login_C4DT_web* - defines which users are allowed to visit the restricted pages in the c4dt site. The logins are done anonymously, all users are mapped to only one web-user
    b. *Login_C4DT_matrix* - defines which users are allowed to use the SSO of matrix to login. The logins are pseudonymous using parts of the credential-id of the user
- *Group*, federating users together, and giving the possibility of handling users in a decentralized way. In the current demonstrator, each industrial partner will be in a group darc.

This setup allows to delegate the responsibility of creating and managing users to the partners, while giving C4DT control to whom it allows to access its resources.
A typical setup looks like this:

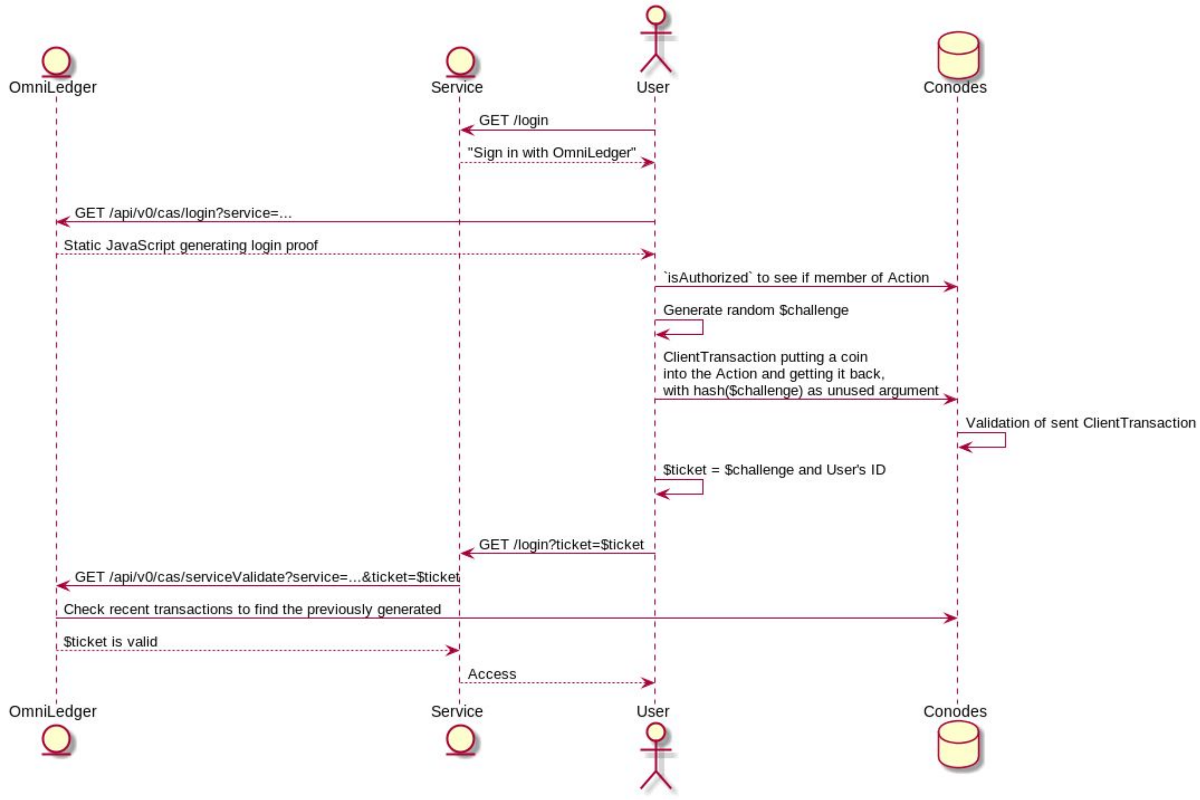

User: AdminC4DT
ID: credential_id
Signer: darc_signer_id
Controlled by
Creates
Darc: Login_C4DT_web
_sign = partner1 | parther2
Darc: Login_C4DT_matrix
_sign = partner1 | parther2
Darc: partner1
Invoke:darc.evolve = partner1_admin
_sign: partner1_admin | partner1_user
Darc: partner2
Invoke:darc.evolve = user_2_admin
_sign: user_2_admin | user_2_2
User: partner1_admin
Users left: 19
User: partner1_user
Users left: 0
User: user_2_2
User: user_2_admin

CAS Login with OmniLedger Credentials

CAS technical details are available in the repo.

First, let it be noted that when creating an Action, a CoinInstance is also created to be able to CAS-login it. This CoinInstance is used to generate a login proof.

The interaction between the user, the service, and the blockchain is shown in the following figure:


OmniLedger
Service
User
Conodes
GET /login
"Sign in with OmniLedger"
GET /api/v0/cas/login?service=...
Static JavaScript generating login proof
`isAuthorized` to see if member of Action
Generate random $challenge
ClientTransaction putting a coin
into the Action and getting it back,
with hash($challenge) as unused argument
Validation of sent ClientTransaction
$ticket = $challenge and User's ID
GET /login?ticket=$ticket
GET /api/v0/cas/serviceValidate?service=...&ticket=$ticket
Check recent transactions to find the previously generated
$ticket is valid
Access
OmniLedger
Service
User
Conodes

1. The user connects to the wanted service and gets redirected to the login javascript
2. The web-client calls `ByzCoinRPC.checkAuthorization` to see if the user is part of the given Action. It is used as a preamble to have a fast way of showing if the user is able to login or not. As the node might be dishonest, this call is insecure
3. The user generates a random challenge, hashes it to the hashedChallenge and puts the hashedChallenge in an argument of a coin transaction from the user's coin-instance to the Action's coin-instance and back, so that the hashedChallenge will be stored on the blockchain
4. ByzCoin verifies that the transaction is valid by verifying that the user has the right to call the `_sign` rule on the Action
5. The user sends its challenge and his userID to the service. Only the user can know the original challenge, because the hashedChallenge cannot be reversed
6. The CAS server looks at the recently added transactions and searches for a transaction matching this public challenge. Then it verifies that
    a. the given userID matches the coin-instance used in the transaction

    b. that the public challenge is indeed the hash of the challenge

7. If there is no error in step 6, access is granted.

For more details how the CAS server verifies the validity of the login-ticket, see https://github.com/c4dt/omniledger/blob/master/webapp/cas/service_validate.go

Note: the login ticket is the concatenation of the UserID and the challenge, as we need to find the user's CoinInstance and its alias.

Appendix A - Details of Contracts

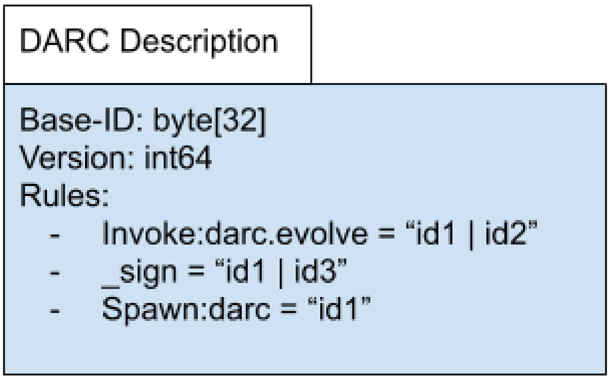
This section holds more details about the contracts used.

Distributed Access Right Control

A central part of how ByzCoin handles the access rights are the DARCs. A darc is described in the paper “Chainiac” from Nikitin Kirill et al. A darc

- is tied to one or more resources and holds rules that define who is allowed to access these resources
- can be evolved, meaning that the rules can be changed. Every time a darc evolves, the darc version is increased by one
- has an identity that is calculated by taking the hash of version 0
- Exists in offline mode (as described in the paper) and online mode (as used in byzcoin), the difference being how to prove what the latest version of the darc is

The following figure gives an overview of a darc:



DARC Description

Base-ID: byte[32]
Version: int64
Rules:

- Invoke:darc.evolve = “id1 | id2”
- _sign = “id1 | id3”
- Spawn:darc = “id1”

The rules for darcs on byzcoin are of the following format:
(invoke|spawn|delete):contractID(\.command)?
A special rule named *_sign* is the delegation rule, that allows a darc to be used in another context.
Every rule is tied to an *expression* made out of *identities* and *operators*. An identity is either a public key (currently only ed25519 and p256 are supported) or a darc. If the identity is a darc, then the *_sign* rule of that darc is used to evaluate the expression.
The operators of the expression are:

- | for ORing identities together (either identity 1 OR identity 2 must sign)
- & for ANDing identities together (identity 1 AND identity 2 must sign)
- () for grouping operators

Future operators should include:

- Thresh(n, id1, id2, …) for defining threshold signing

Restricted and Unrestricted Darcs

For security reasons, normal, restricted darcs are not allowed to add new rules. This is to avoid that a user can evolve a darc and be able to spawn instances without the consent of the admin.

Unrestricted darcs are allowed to add new rules.
The two darcs are implemented as two different contracts.

Spawner Contract

The byzcoin system depends mainly on darcs to define who is allowed to create instances. For a general user handling, this is not useful, as we want to limit the possibilities of users to create new instances. If every user could create an infinite amount of instances, the system would very quickly be flooded. Using the available coins in byzcoin, a spawner contract is defined by the main admin, which can be used to spawn new instances.
A Transaction that wants to spawn a new instance will have at least two instructions:

1. Fetch coins - remove coins from a coin instance and put it on the stack
2. Spawn instance - call the spawner contract, which will consume the coins and create the instance

The second step can be repeated in the same transaction, as long as there are enough coins on the stack. The following costs are defined in the spawner contract:

- Darc
- Coin
- Credential
- CalypsoWrite
- PopParty
- RoPaSci (Rock Paper Scissors)

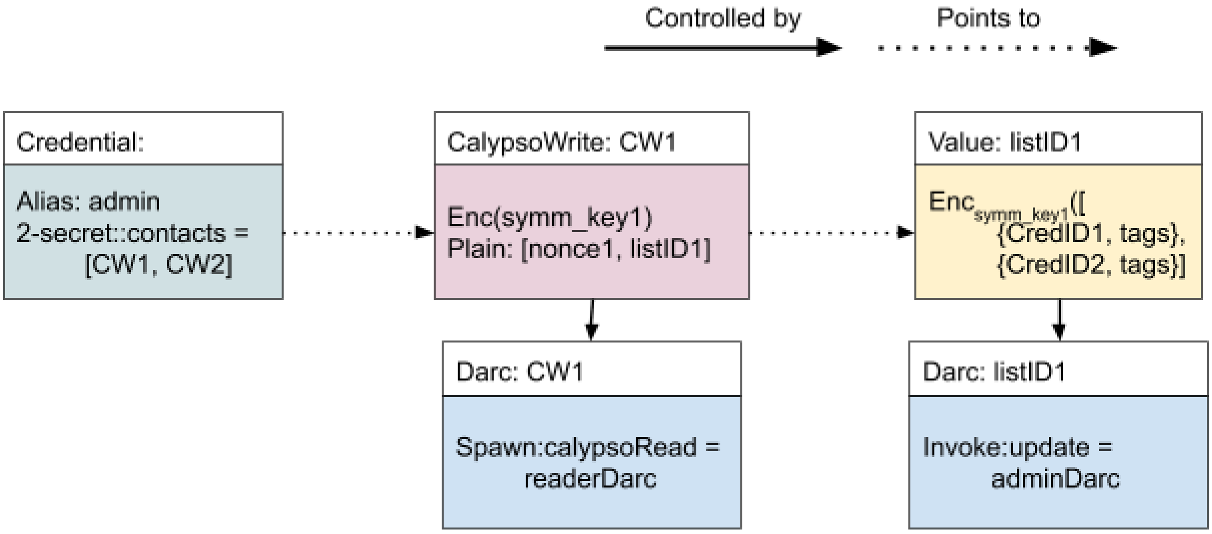
A special mention is the CalypsoWrite contract, as it must allow the spawning of CalypsoRead instances. To allow this, the CalypsoWrite instance also has a ‘cost’ field that defines how many coins must be available before spawning a new CalypsoRead instance.

Appendix B - TODO List

Proposed Extension

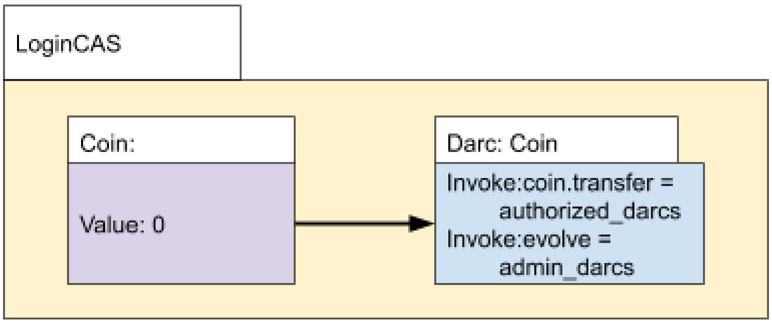
Problem: C4DT uses the omniledger-demonstrator that incorporates a user-management system. As more and more users are signed up to the system, it gets difficult to manage them, even if we suppose that the users are handled by the affiliated partners and labs.

Proposed solution: Create a directory service that allows admins to share user-IDs and some tags attached to it, so that whatever one admin does (adding, removing, modifying) can be viewed by another admin.



CAS Login

The CAS Login should be identified by the ID of the coin-instance, as the coin-instance points to the DARC:



Phonebook extension

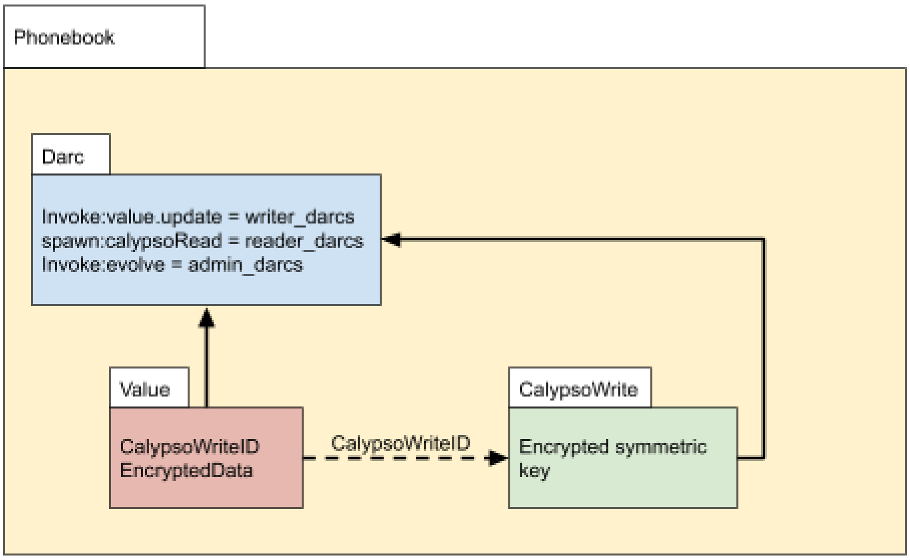
The first version of the omniledger interface had only a list of contacts, DARCs, and actions, but there are a few problems with that:

- As more contacts get added, the list becomes long
- All items are loaded when connecting to omniledger, which takes a long time
- If you want to link to a contact, you need to ask the creator of the contact for the ID

To solve these problems, here comes the phonebook! It has the following properties:

-
- It is stored encrypted on the blockchain, using calypso
- It contains a list of IDs that point to one of
    - Credential
    - DARC (for groups)
    - Coin (for LoginCAS)
    - Phonebook
- When creating a user, a Phonebook is also created and linked in its credentials
- The UI keeps a cache of all phonebook entries as a `map<ID, PhonebookElement>`
  It is used as a local cache of the actual phonebook entries. `PhonebookElement` is a structure of
    - `Description string`
    - `Type oneof(credential, darc, login, phonebook)`
- On startup, the UI updates the cache by checking whether there is a change in any of the known phonebooks. If so, the changed phonebook is read and the internal cache (stored in storageDB) is updated.

The phonebook points to a value-instance and has at least 3 instances:



Creation

Upon creation of a phonebook, the user has to give the DARCs that define the writers, readers, and admins to the phonebook.