Implementation of an intuitive and insightful blockchain explorer

Julien von Felten

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Optional Master Semester Project

June 2020

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Noémien Kocher
EPFL / DEDIS

Contents

1 Introduction

1.1 Project

Since the development of the first blockchain in 2008 by Satoshi Nakamoto[1], it has been quite hard to understand and to have a good visual representation of what a blockchain is. Various blockchains have been developed but it remains not user-friendly to verify the content of a blockchain, to see how the different intrinseque mechanisms of the blockchain work such as the content of each cryptographically linked block and how they participate in evolving the state of the blockchain.

The goal of this project is to implement a tool that can parse and display insightful information about the DEDIS blockchain called ByzCoin. It aims at prodiving an easy way for users to see and understand what is inside the blockchain by offering an intuitive and powerful user interface.

This project is developed by two students(Anthony Iozzia and Julien von Felten - myself) with distinct parts in order to have two reports for each part. This report is about the second part of project. The tool could not be finished on time because of an extraordinary event which is the quarantine due to the COVID-19. The EPFL decided to cancel all the students' projects for two weeks and I was sick for two weeks. Due to these circumstances, the project fell behind. It has been decided to focus on having a tool that works even though it is not an intuitive and insightful interface. Most of the mechanisms needed for this tool have been developed, but some improvements of the interface still remain to be developed.

Three main features are developed in this project:

1. The visualization of the blockchain with the different blocks.

2. The detail of one block with all transactions, instructions, verifiers, back and forward links.

3. The evolution of one instance through the whole blockchain.

Anthony is responsible for the first feature whereas I am responsible for the second and third part. In each part, the creation of an intuitive interface is implicit. The initial split was different but because of the situation and because Anthony fell behind, I implemented the two features. Hence, I could not do all the planned tasks in order to be sure that the three main features would be implemented and would work.

[1] https://en.wikipedia.org/wiki/Blockchain

1.2 Background

ByzCoin[2] is a distributed ledger that implements the protocol described in the OmniLedger Paper[3]. It is part of a bigger project which is called Cothority[4]. Cothority is a framework related to large-scale collective authorities (cothorities), which distribute trust among a number of independent parties to allow scalable, self-organizing communities. Most of the projects are using the blockchain called SkipChain which is part of the cothority. A skipchain can be seen the same way as a blockchain, however it allows to skip blocks forward and backward in opposition to the blockchain. ByzCoin is composed of many blocks, which themselves are composed of a variable number of transactions. Each transaction contains instructions which themselves contain directives that manipulate an instance of a smart contract. The instance of a smart contract can evolve in time with the executions of instructions with different state: spawn, invoke and delete. The third main feature of the project is to be able to follow the evolution of one smart contract instance throughout the blockchain.

In order to be able to make request to the skipchain of ByzCoin, it is required to use the Cothority client library in Javascript. A structure PaginateRequest is sent to the skipchain and a structure PaginateResponse[5] is received with the result of the request. These calls are made asynchronously using a library called rxjs[6]. This kind of programming is called "event-driven programming", more specifically in this project, "reactive programming". This library provides more flexibility than the promise of JavaScript. Rxjs is needed since, in the client library, PaginateRequest returns an Observable of SkipBlocks. This project uses Observables and Subjects of rxjs. The connection with the skipchain is done using a WebSocketConnection[7] which is a single peer connection to one single node and a WebSocketAdapter[8] which is an adapter to use any kind of websocket and interface with a browser compatible type of websocket. To find the server identity, a Roster[9] class is used which is simply a list of server identities. The roster is given as parameter to create a WebSocketConnection. s

[2] https://github.com/dedis/cothority/tree/master/byzcoin

[3] https://eprint.iacr.org/2017/406.pdf

[4] https://github.com/dedis/cothority/

[5] https://github.com/dedis/cothority/blob/master/external/js/cothority/src/byzcoin/proto/stream.ts

[6] https://rxjs-dev.firebaseapp.com/guide/overview

[7] https://github.com/dedis/cothority/blob/master/external/js/cothority/src/network/websocket.ts

[8] https://github.com/dedis/cothority/blob/master/external/js/cothority/src/network/websocket-adapter.ts

[9] https://github.com/dedis/cothority/blob/master/external/js/cothority/src/network/proto.ts

1.3 Subjects and Observables

The Observable represents the idea of an invokable collection of future values or events. The Subject is an even emitter, which multicasts a value or an event to multiple Observables. As it is written in rxjs[10]: "A Subject is like an Observable, but can multicast to many Observers. Subjects are like EventEmitters: they maintain a registry of many listeners". Hence both of them are used in the same way, but Subjects can have more than one subscriber. They are used instead of promises because of the capability to push multiple values whereas promises push only a value once. When creating a Subject or Observable, the function subscribe has to be defined in case of the event *complete*, *next* and *error* with some parameters. Then, when one of these events is called, the code defined in this event will be triggered. It can be done using the code `Observable.next(parameter)` for example. So an event can be defined and triggered only when desired, thanks to the execution function.

1.4 D3

The library that is used for the visualization is the well known D3 libary[11]. It is a library widely used to modify SVG, HTML and CSS using JavaScript. The function `d3.select()` can be used to select a DOM element, `d3.append()` to append a child to the selected element, `d3.attr()` to change or set an attribute of the selection (for example the class of the element) and `d3.on()` to set different action depending on the event such as click or mouse over. Many other functions are available but these are the most used ones in this project.

1.5 Actual tools

This new tool will be able to replace three tools:

1. Status Explorer[12]
2. Columbus CLI[13]
3. bcadmin[14]

The first motivation to create another tool is that it can replace these three. The current three tools have different goals. There is no proper visual tool for the ByzCoin to explain and to show to the DEDIS clients an intuitive representation of the blockchain. It is also hard to verify the

[10] https://rxjs-dev.firebaseapp.com/guide/Subject
[11] https://d3js.org/
[12] http://status.dedis.ch/#/
[13] https://wookiee.ch/columbus/
[14] https://github.com/dedis/cothority/tree/master/byzcoin/bcadmin

content of each block and follow the evolution of one instance using these tools. They present also some performance and fluidity issues. This is why the new columbus project called Columbus makes sense: create a better interface to understand the blockchain for a person who has no previous knowledge about blockchain and the possibility to quickly verify the content of a block or show the evolution of a smart contract instance for the users of the blockchain.

The Columbus CLI project is quite close to the second main feature of the project, hence it is taken as a basis for the code of this part of the project.

2 Design

During the first seven weeks of the project, it was split into two different Git repositories with different goals: the visualization of the blockchain (first feature) and the visualization of the inside of a block with the evolution of an instance ID (second and third features). This report is about the second part of the project. At the beginning, the mechanism for the browsing of the blockchain was developed, which is the implementation of the third feature[15].

After the first weeks, both parts were merged into one Git repository in order to have one project. Since there were two different parts without any link to one another, a third part was implemented. It is the visualization of the details of the clicked block which will link the blockchain to the evolution of one instance. In this report, we will talk about the mechanism of browsing the blockchain, the display of the information about one block and the result of the browsing which will show the evolution of one instance.

The fact that the project fell behind imposed us to keep the code clean and to keep the best practice using Git. This effort is not wasted as other students might continue this project. This clean code will allow them to understand the project much easier.

2.1 Browsing of the blockchain

The Columbus CLI tool allows users to go from the first to the last block of the blockchain and displays the content of each block using simple clicks. The Columbus CLI was adapted in this project in order to be able to fetch blocks with a click. It then needs to be adapted using Subjects to be able to go from the first to the last block without any human action.

Instead of fetching only one block after another, the blockchain API allows to fetch $pageNumber * numberPage$ blocks at a time. An error occurs when there are less blocks remaining than the requested ($pageNumber *$

[15] https://github.com/dedis/student_20_columbus_vue/

numberPage) blocks at the end of the blockchain. This ends the browsing. Hence, to avoid this problem, it was decided to implement the browse function in an recursive manner: the function is recalled using $pageNumber = numberPage = 1$ in order to fetch all the blocks.

Every time a block is fetched, it is parsed to check whether there is an instruction that contains the searched smart contract instance which, in this case, is added to a list to be displayed to the user later on. The filter was done using the instanceID of an instance. This will allow to return the list of all blocks that contains this smart contract instance and all its instances.

There exist four functions for the browsing:

1. `function getInstructionSubject(Instruction)`, initializes all the variables needed for the browsing and calls the browsing function.

2. `function browse(number, number, string, Subject, Subject, SkipBLock[], Instruction[])`, starts the browsing until it gets an error, then calls itself recursively.

3. `function getNextBlocks(string, number, number, Subject, Subject)`, gets the next blocks.

4. `function handlePageResponse(paginateResponse, WebSocket-Adapter, Subject, Subject)`, handles the paginateResponse of the request.

Another function is added in the project to notify a Subject of the progression of the browsing which will be used to have different feedback for the user.

Initially, the display of the evolution was done in the Browsing class, but it was then moved to a new class that only focuses on the display of everything.

2.2 Displaying information

After the first seven weeks, both parts of the project were merged together without any big issues since the parts were completly independent and without any conflict. A new class, `DetailBlock`, has to be developed, which links both parts of the project. This class creates the interface of the details and of the evolution of a block, it displays all the transactions, verifiers, back and forward links in one container and all the instances found related to a browsing inside another container. The Browsing class has been changed in order to be responsible for only browsing and notifying its progress without any responsibility of modifying the view. The DetailBlock class is now responsible for displaying the progress bar. This choice was not the easiest and fastest to implement since it was initially the Browsing class that was

implementing it. It was decided to move it from the Browsing to the DetailBlock class because the latter is the class that displays everything related to a block. Even though it took time to change from the first implementation to the second, it keeps a better architecture for the code since one class is responsible for updating the view while the other one is responsible for providing functions to retrieve information about the blockchain.

Since the project will not be finished, it has been decided to put an emphasis on keeping a good code architecture in order to improve its maintainability and to facilitate the implementation of future features. This choice creates a clearer code since the project might be continued by a third student. It was also decided to use tslint and prettier to check readability, maintainability and functionality errors.

The class DetailBlock has two main functions:

1. `function listTransaction(SkipBlock)`, displays the content of the clicked block.

2. `function printDataBrowsing([SkipBlock[], Instruction[]])`, displays the result of the evolution of one instance.

There is also a function that highlights the blocks in which an instance of the evolution has been found. Different functions are also present to handle the progress bar with a loading screen.

2.3 Utilities

Different classes were implemented for the utility:

- `Flash`, handling the display of the flashes.

- `TotalBlock`, getting the last block of the blockchain.

- `Utils`, containing different utility functions.

The Flash class is displaying a flash of different colors depending on the error code with a text which can explain the problem or the information to the user. This allows to highlight for the user what mistake was made.

The class that gets the last block was initially designed to get the total number of blocks for a feedback for the user in the loading screen. It was then realised that returning the last block instead of a number would be more practical for the long term of this project. We could imagine that the Browsing would be optimized using a two-way Browsing starting from the first and last blocks and stops when they meet each other in the middle of the blockchain. The class is also adapted to avoid going through the whole blockchain more than once in case the function is called more than once. It keeps the last block seen at the previous run in order to start from there

and check whether other blocks were appended. This makes the first call to be quite long, but then the next calls are sped up.

The utility file is mostly for the other student.

3 Implementation

The code of this whole project has:

- Three founding classes, which can be seen as the three main features of the project:
  1. `BlocksDiagram`, implementes the visualization of the blockchain.
  2. `DetailBlock`, shows the content of a block and the result of the browsing for the evolution of one specified instance ID.
  3. `Browsing`, finds the evolution of one specified instance ID.
- four utility classes
  1. `Flash`, displays a flash box.
  2. `TotalBlock`, get the last block of the blockchain.
  3. `Utils`, different utility functions.
  4. `Roster`, a list of server identities.
- one main class
  1. `Index`, the main class that initializes all classes.

The execution graph of the project can be seen at Figure 1 which starts when the webpage is loaded.

3.1 Browsing mechanism

As already explained in the design section, the browse mechanism uses the recursivity to cope with the error returned by the PaginateResponse that happens when the number of blocks supposed to be fetched is greater than the remaining blocks. In this case, the browsing is called again with the parameters $pageSize = numPages = 1$. For example, if the number of requested blocks is 10 in the next request and the number of remaining blocks is 9 (this number is not known in advance), an error occurs and the 9 blocks are not returned. The browsing restarts from this point making 9 requests of one block each in order to reach the end of the blockchain.

The browsing is started from the class DetailBlock which calls the function `getInstructionSubject(Instruction)` of the file browsing.ts whenever the user clicks on the button to search for an ID. This function initializes different global variables in order to start a browsing. It initializes the two

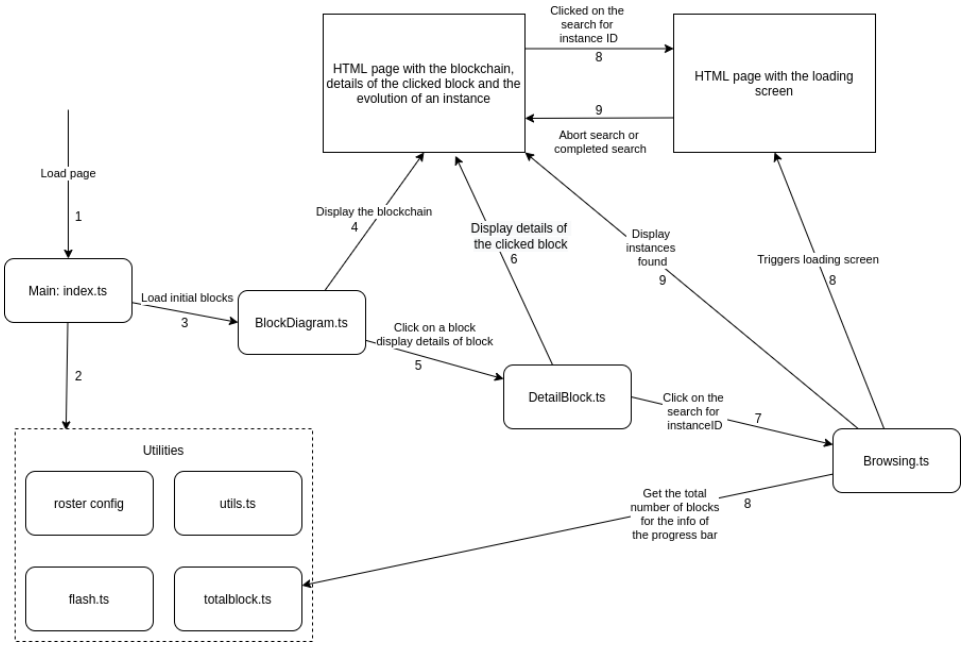

Figure 1: graph of the execution of the project

return values which are both Subjects: the first Subject is used to keep track of all the blocks and instructions that are found that belong to the evolution of the clicked instruction. The second Subject is used in order to update the loading screen, hence it returns an array of number which will be discussed later. In this function, the Subject for the class TotalBlock is initialized in order to get the number of blocks in the blockchain. This will set the variable $totalBlockNumber$ which is used for the loading screen. The contract ID is also set to make a comparison with each instruction of the blockchain. This contract ID can be found in the $instruction.instanceID$ given in parameter.

The browse function creates a Subject with an array of two values: one number and an array of SkipBlocks. This array will be used after the asynchronous call of the requested blocks to trigger the function *complete*, *error* or *next* of this Subject. These three events are defined as follows: the event *complete* notifies with a flash the end of the browsing and notifies the values found for the requested instruction. The event *error* checks which error occured and depending on the error, triggers another browse with different parameters as explained above, or it displays a flash with an error message. The event *next* is the most complex one. It gets as parameter an array with two values: the index of the requested block and the last fetched SkipBlock. It checks if this block has one instance of the requested instruction inside and if it is the case, this block is added to the structures which will be returned to the Subject for this instruction. Then it checks if this block is the

last block of the last page requested of one request (recall that the number of block requested is $pageSize * numPages$). If it is the case, it will call the function `getNextBlocks()` which will send another request to get more blocks with the ID of the last block found. In the case that the last block of a request has no more forward links, it means that we reached the end of the blockchain. The reseach can be cancelled also by setting the variable `abort = true`. This will complete the Subjects in order to get the result already found in case of an abort.

The function `getNextBlocks()` requests the next $pageSize * numPages$ blocks by creating a paginateRequest which returns an Observable of blocks. Hence it remains to set what will happen in case of a *complete* (nothing), an *error* (an error occured because of the connection so the webSocketAdapter is set to null) and the *next* which will call the function `handlePageResponse()` with the data received. If this function returns the number 1, it means that the error of "too many blocks are requested" occured. It will trigger the event *error* of the browse Subject.

The last function that is useful for the browse mechanism is the function `handlePageResponse()` function which gets the blocks out of the data from the paginateResponse and triggers 3 actions: the first is to increment the number of seen blocks, the second is to call the `seenBlocksNotify()` function and the third is to call the event *next* from the browse Subject. If no error occurs, it returns 0, otherwise it returns 1.

To sum up these 3 functions, the first function `browse()` defines what has to be done with a block and calls to get the next blocks. The function `getNextBlocks()` is responsible to get the next $pageSize * numPages$ blocks and calls `handlePageResponse()` to handle the paginateResponse which will call the browse event *next* for each block.

The function `seenBlocksNotify()` uses a "blackmagic" condition:

$$seenBlocks \,\%\tilde{}\tilde{}(0.01 * totatBlockNumber) == 0$$

It will be true for each rounded percent which will trigger the event *next* of the Subject progress with an array of numbers. The numbers in this array are the percentage accomplished of the browsing, the number of blocks seen, the total number of blocks and the number of instances found. This Subject will be used for the loading screen. In case the $totalBlockNumber$ is still not updated, it then notifies at each seen block and this spam is handled in the class where the loading screen is updated.

There are many Subjects in this class, but if someone wants to use it, it can call the function `browsing.getInstructionSubject(Instruction)` and it will return 2 Subjects: one is the result of the evolution of the instruction with the SkipBlocks containing this instance and their instruction, the second one is the Subject to notify the progress of the research which allows to have an estimation of the time remaining and a feedback for the

user.

3.2 Displaying information

To display the different information, there are two main functions in the file detailBlock.ts: the function `listTransaction(SkipBlock)` which displays the content of the clicked block and the function `printDataBrowsing()` which displays the result of the browsing. There are some utility functions such as creating and removing highlights of some blocks, creating, updating and removing the loading screen. The functions to create and remove the highlights are quite simple: they take as parameter the list of SkipBlocks, it goes through the list and add or remove the proprety *stroke* using `d3.attr("stroke", "color")` and `d3.attr("stroke-width", "size")`. It also sets an event on mouse over that makes the stroke wider.

The first main function is triggered when a click on a SVG block happens. Using the Observable *skipBclickedObs* from the constructor, it subscribes to the event *next* which will call the `listTransaction(SkipBlock)` function. In this case, it will display the content of the block. It will change the color of the clicked block and starts building the different elements. Using getuikit[16], which is a lightweight and modular front-end framework for CSS, it creates a container for each transaction and for the block details (such as verifiers, back and forward links). The *accordion* class from getuikit is used for each detail that can be expanded with more content. In each transaction container, there is one instruction container for each instruction which holds the information about this instruction and the button to start searching the evolution of this instance. Hence, all transactions and all instructions of each transaction are displayed with their types, their arguments and their information.

The biggest challenge for this class was to know how much information to display. It is not a trivial question since too much information will make the user give up searching for it whereas not enough information will not be useful at all. This is why the help of potential users is greatful. This will be discussed later.

When a click on the search button and a confirmation from the confirm box happen, a variable is created to get the two Subjects from the function `browsing.getInstructionSubject(Instruction)`. The first Subject defines the event *next* as simply starting the function `printDataBrowsing ([SkipBlock[], Instruction[]])`. The second Subject defines the event *next* as updating the loading screen with the function `updateLoadingScreen (number[])` and the event *complete* as the end of the browsing which removes the loading screen with the function `doneLoading()`.

When `printDataBrowsing([SkipBlock[], Instruction[]])` is called,

[16] https://getuikit.com/

it displays all the information about the evolution found by the Browsing class. It will also add some highlights to the concerned blocks. The structure of the function is close to the `listTransaction(SkipBlock)` function using *accordion* from getuikit.

To create the loading screen, the function `createLoadingScreen()` creates a load container which is placed above everything. It displays the DEDIS logo with a link to the website of DEDIS[17], a spinner, a progress bar and a button to abort the research. This button has a on click listener which will change the variable `browsing.abort` to true to interrupt it. To update the progress bar, the second Subject from the Browsing class returns an array of numbers. The function `updateLoadingScreen(number[])` updates the values of the progress bar from what it has been notified of. Finally when the loading is done, the load container is simply removed with the function `doneLoading()`.

3.3 Flash feedback

To have different feedbacks for the user, a flash can be displayed. The class Flash from the file flash.ts handles the display. An enumeration is created with different Flash types: ERROR, WARNING, INFO, OTHER. OTHER is kept for maintainability of the code if in the futur, more classes have to be added. The public function is `display(Flash.flashType, string)` which parses the flashType. Depending on the type, it displays a different message with a different color. An event on click is created to remove the flash in case of a click on the cross. It results in a pop up that can be removed at anytime.

3.4 Number of blocks

The file totalBlock.ts has been created in order to get the last block of the blockchain. It was initially designed to get the index of the last block, but it would be more pratical to return the last block in case this block is needed in the future. This class has one public function `getTotalBlock()` which returns an Observable of a SkipBlock. It then calls the private function `getLatestBlock(string, Roster)`. This latter function has the same structure than the function `browsing.getNextBlocks()` but selects the last forward link (the furthest) until the last block is recovered. In this case, it notifies the Observable with the last block and also updates the hash of the starting point for the next use. This avoids starting all the time from the first block and going through the whole blockchain. From the second time it is called, it only checks whether one or more blocks were appended.

[17] https://www.epfl.ch/labs/dedis/

3.5 The main

The main function is called `index.sayHi()` which is loaded when the user opens the page. This function creates the utility classes such as Roster, Flash, TotalBlock and the founding classes such as BlocksDiagram, Browsing and DetailBlock. This class is the starting point of the whole project.

4 Interface

The interface of the project will be presented with different screen shots an an explanation of each part, however only this part of the project will be detailed. Later a user exerience will be discussed with the observations and the suggestions to have a more intuitive tool. The biggest problem with this project is that it happens during the COVID-19 outbreak and the sanitary containment. The project fell behind and it was decided to focus on the backend and if the time could permit, to focus on an intuitive interface. Unfortunately due to lack of time, the user experiment was made the last week of the project which will show improvement for the future student.

4.1 UI overview

The Figure 2, 3 and 4 show the different steps a user can perform with this tool. Initially, the user does not see any details about any block. They can drag left and right, zoom in and out the blockchain. They can perform a click on a block to display the details about this block. Then, when expanding one instruction, the button to search for the evolution can be seen. While searching for the evolution, a loading screen is created for feedback. At the end of this search or if it is aborted, the result of the evolution is displayed on the right side. Every line that has a '+', can be expanded to have more details.

In the Figure 2, it is shown what a user sees when he clicks on the blue index 4 block. Here is an explanation of each number:

1. The Columbus title with a description that spawns when the user performs a mouse over.

2. The blockchain with the blocks from index 4 to 13. It is possible to drag left and right, zoom in and out to go to other blocks. Each block is clickable to display its content in the next point (3). When a click is performed on a block, this block becomes darker (it is the case for the block 4 here).

3. The content of the clicked block with different information: the index of the block, its hash, the list of all transactions contained in this block and some more details about the block such as the verifiers, the back and forward links.

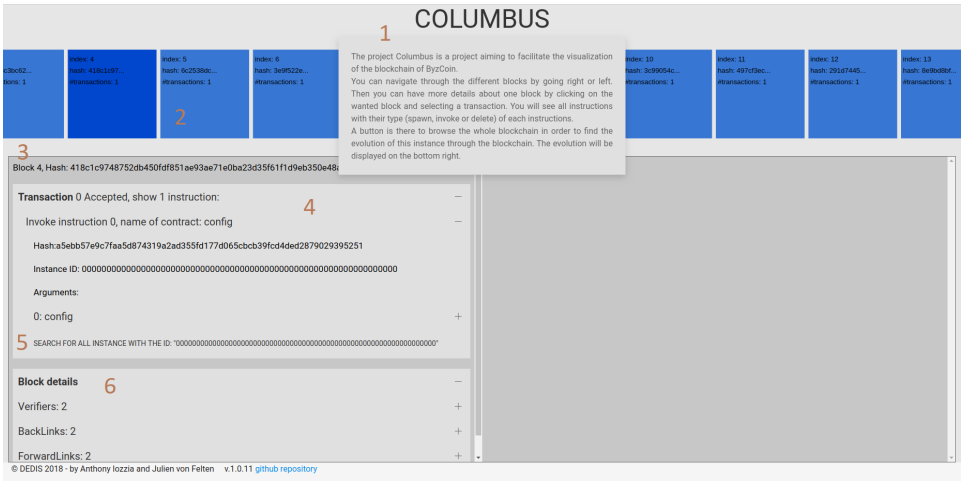

Figure 2: Screen shot of the content of the index 4 block

4. The details of one transaction which show the instructions contained in this transaction (here there is only one). The details of the instruction are also shown with the hash, the instance ID and the argument names. The arguments can be clicked to see their value.

5. The button to start the browsing of the blockchain in order to find the evolution of the instance ID (Here: 000[...]000). In case of a click on this button, a confirmation window appears to be sure that the user really wants to perform this operation.

6. The details of this block such as the verifiers, the back and forward links. Each of these three can be expanded to see more details.

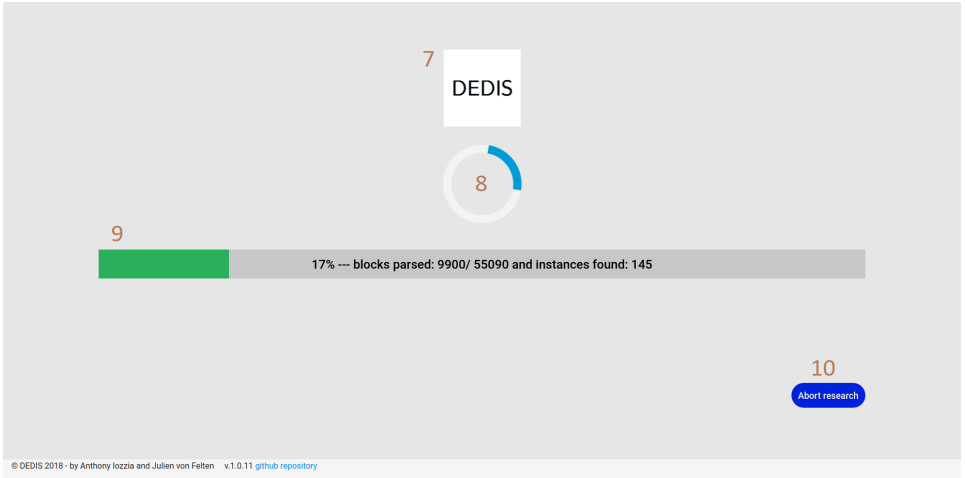

Figure 3: Screen shot of the loading screen for the research of the evolution of an instance

The Figure 3 shows the loading screen after the confirmation for the browse. There are four different elements in this screen:

7. The DEDIS logo that is clickable to open their website.
8. A spinner to catch the eye of the user while this long process of searching is happening.
9. The progress bar in order to have a feedback for the user. The green bar goes from left to right according to the percentage. It shows the number of parsed blocks, the number of total blocks and the number of instances found.
10. The abort button in case the user wants to stop the search.

In the Figure 4, it shows the result of the browsing. It happens after the loading screen. The left part remains the same whereas the right part shows the evolution of the instance. Here is the explanation for this screen shot:

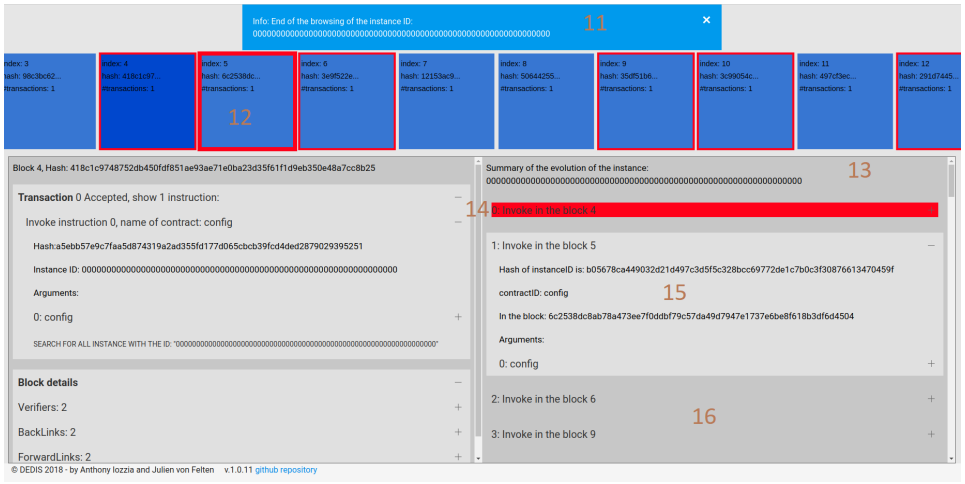

Figure 4: Screen of the content of a block and the evolution of the researched instance with the ID 000[..]000

11. The Flash with the type "INFO" which notifies the end of the browsing with the instance ID. It can be removed with the cross.
12. Some blocks have been highlighted. These blocks are the ones that are found in the evolution of this instance. We can see that the index 5 block has a bigger highlight, this is because the user is currently performing a mouse over for the invoke in the block 5 in the evolution container.
13. The container of the evolution. It shows all the instances found with the index of each block. Each instance can be expanded to see more details.

14. The research started from the block 4 which explains why the background is red.

15. The details of one instance. There are the hash of the instance, the name of the contract, the hash of the block this instance is in and the arguments.

16. The list of the evolution of the searched instance. It can be scrolled to show more instance from block index greater than 9.

4.2 User experiment

A short user experiment was performed in order to have some feedback as a starting point for the next student that will continue this project. The user experiment could not be done early enough due to the reasons stated previously. Hence, the chosen users are from my family and friends which were available. They were a great help even though the user experiment should not be conducted with friends and family for the reason that they might not be honest. It was also a logistic and timing issue due to the situation.

There are three types of users for this experiment: the technical user that knows what a blockchain is and will be using the tool alone, a non-technical user with basic knowledge about the blockchain that wants to know more about the blockchain and will be using the tool alone, a non-technical user that knows nothing about blockchain and is guided by a technical user (in this case, I did the technical user to guide her). One user from each group was taken to conduct this experiment. The experiment started by a short introduction about the SkipChain and ByzCoin, then an introduction about the project with the different goals for this tool.

Then 11 tasks were asked:

a. Find and read the description of the project.

b. Find block number 64, how many transactions are in this block?

c. Browse this block.

d. Find block number 4.

e. Search for the evolution of Instance ID: 000[...]000.

f. Stop after 15%.

g. Why the block number 4 is darker?

h. Why is there a red stroke over some blocks?

i. Why is the line ”0:Invoke in the block 4” highlighted in red?

j. What do you observe when you put your mouse over the evolution?

k. Look at the argument of the second invoke in block 6.

The users were encouraged to talk about what they are doing, searching and expecting. Instead of explaining the result of each user, all the results are mixed together since they do not diverge a lot even though their background and knowledge about the blockchain is different. Also, the feedback is for the whole project and not only this part.

For the first task, it was quite hard for them to find the description. It would be more intuitive to have a small button (information icon) and when the user clicks on this icon, the description shows up. Also, the description should be more precise and clearer for the user. They should understand what they can do by reading this description.

Then, they have to perform a drag from right to left in order to go to the block 64. None of them understands that the blockchain can be dragged, zoomed in and out. They complain about the size of the blocks being too big (they currently occupy 40% of the screen!). This number should be reduced. Two users disliked the navigation from left to right through the blockchain from block 1 to block 64 and go back. They said "it is boring to drag that many times". One of them suggested to have a button "home" to go back to the first block and a button "end" to go to the last block (either last loaded block or last block of the blockchain). The second one suggested some arrows on the left and right that are clickable in order to avoid dragging many times. It would be more intuitive to have a scroll bar below the blockchain to understand that there are more blocks than displayed. This would also help scrolling.

The biggest problem, that was noticed and might be increasing, is the technical knowledge of the user. The two non-technical users asked why the transaction list started from 0 and what the word "parsed" in the loading screen means. This shows the difference between technical and non technical users. This problem can occur also with the description of the project or some other future features. One solution would be to have two different interfaces depending on the user. In this case, different information with the assumed knowledge could be easier to understand. Otherwise, a great care has to be provided about the information displayed and whether they are understandable to all users.

The button to search for an instance ID is not clear at all. It would be better to have it on the right side of the container with a border and a different color.

After the click on the abort button, it takes some times to go back to the home screen. Hence it would be suggested to have a feedback message indicating that the abort command was acknowledged since there can be a delay clicking and the search aborting. Since the users do not know if the abort worked or if it has to be clicked again, they are trying to click again on

the button and are confused while the search seems to be still running. One suggestion would be to change the text of the button from "Abort research" to "Aborting...".

Once the evolution is displayed, all users expected the blockchain to move when clicking on one instance of the evolution. Hence, when clicking on the Invoke from block 4677, they would expect the blockchain to be centered around the block 4677.

The problem for this project was the lack of time because of the situation. Unfortunately, no user's opinion could be taken into account. Now that an interface is working, a bigger user experiment could be conducted before starting the next part in order to define clearly what could be intuitive or not.

4.3 Suggested modifications to the UI

Hence, to sum up with the user experiment and some ideas for the UI, here are some suggested modifications:

1. Create a short tutoriel button which will display some information about different parts of the project.

2. A line from the selected block to the transaction container to show that this information belongs to that block.

3. A problem occurs when the server is not started. This error will never recover even though the server can be started afterwards. It would be great to implement a recovery for this error.

4. Explore and find a block which contains an invoke of invoke and have a thought of how to display this case.

5. Create different icons for the different accordion of CSS: transaction, spawn, invoke, delete, verifiers, back and forward links.

6. A problem occurs when the contractID "860df[...]" is searched since it contains many instances. An idea would be to display only 50 or 100 instances and the possibility to display more depending on actions (click button or scroll to the end).

7. When the window size is smaller than 1000, the two containers for the information are one above the other and they are not scrollable. It would be great to have a scroll bar. Also the instanceID in the flash is displayed outside of the flash container. It would be suggested to display only a part of the instance ID, for readability.

8. Display the first block (index 0).

9. In the code, they are repetitions in the class utils.ts and totalBlock.ts with the function that gets a block. They could be merged and just start from different hash number using the same function.

10. Add an information button to describe the project.

11. Find a different way to navigate through the blockchain: scroll bars, arrows, "home" and "end" button or another intuitive mechanism.

12. Add some keyboard interactions (home button, arrows of the keyboard, zoom with the arrows, etc)

13. Reduce the gap between technical and non-technical user.

14. Create a more visible button to search for the evolution.

15. The abort button should give a feedback to show that it will abort as soon as possible.

16. Center the blockchain around the block with the selected instance in the evolution.

5 Future work

The first element to improve is the suggested modifications mentioned above to apply to the project. In order to be sure that it would be a good idea to have these adjustments, another bigger user experiment could be conducted with more subjects (mostly the people from DEDIS lab and some potential users).

The problem with the search for one instance is the waiting time. It would be a sought after improvement to speed the browse function. Since JavaScript is a single threaded language, the suggestion would be to use workers. One could start from the end of the blockchain moving backward meanwhile the second starts from the beginning moving forward. They stop when they meet each other. This idea would take a lot of time to implement and is not assured that it would result in better execution time.

The possibility to search for a name of a contract with a part of the ID in the blockchain would be a great advantage for the user instead of using the whole ID. Since it is a long string, it is hard to remember it, whereas the name of a contract is usually 8-10 letters of an english word and a small part of the ID. A text input could be created near the blockchain and the user could write the contract name with the beginning of the ID and the browse would start.

6 Conclusion

This project shows the importance of a good and intuitive interface. The idea of a software and how it is implemented can be the best ever, if the interface is not appealing and user-friendly, nobody will be willing to use it. For example, Zoom, the software for the visioconference probably sells the data to other companies but everyone is willing to use it because the interface is appealing, it is really useful in this situation with the containment, easy to use and to understand. Hence it shows the important of creating a great interface that will appeal the users, this is done by developping the UI with the user and not only for the user. Thus it takes more time to develop the project initially but it saves much more time later on when modifications happen.

The code of the interface can also be done without much architecure or any previous thought, but it can be messy and even impossible to add some small changes. It takes more time to implement a modular code initially but it will be easier to maintain and to modify details. It is also challenging to implement a project with another person to coordinate who will be doing which part, which architecure to use, what to start with. Everyone thinks differently and even after discussion, it can be different than expected because the other student understood it differently. It is also a great experience to use Git for a group project.

Even though this project is not finished, it is a tool that could be used and a great foundation to create the perfect visualization tool for the blockchain.

I hope that you will enjoy using it and that it will be improved since no other software exists to visualize blockchains intuitively. I really enjoyed this project as it taught me many different aspects of the event-driven programming and how to communicate with a blockchain. I am thankful to Noémien Kocher which was my supervisor, he was patient, took the time for all questions I had during the project, explained to me many aspects of good software development and architecture. I am also thankful to the DEDIS Lab team who accepted me for this project. The repository can be found with this link[18].

[18] https://github.com/dedis/columbus-united

References

[1] https://en.wikipedia.org/wiki/Blockchain

[2] https://github.com/dedis/cothority/tree/master/byzcoin

[3] https://eprint.iacr.org/2017/406.pdf

[4] https://github.com/dedis/cothority/

[5] https://github.com/dedis/cothority/blob/master/external/js/cothority/src/byzcoin/proto/stream.ts

[6] https://rxjs-dev.firebaseapp.com/guide/overview

[7] https://github.com/dedis/cothority/blob/master/external/js/cothority/src/network/websocket.ts

[8] https://github.com/dedis/cothority/blob/master/external/js/cothority/src/network/websocket-adapter.ts

[9] https://github.com/dedis/cothority/blob/master/external/js/cothority/src/network/proto.ts

[10] https://rxjs-dev.firebaseapp.com/guide/Subject

[11] https://d3js.org/

[12] http://status.dedis.ch/#/

[13] https://wookiee.ch/columbus/

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

[15] https://github.com/dedis/student_20_columbus_vue/

[16] https://getuikit.com/

[17] https://www.epfl.ch/labs/dedis/

[18] https://github.com/dedis/columbus-united

[19] https://bl.ocks.org/

[20] https://typedoc.org/

[21] https://www.nngroup.com/articles/#videos