

Building an HTML/JS app to visualize the contents of a Skipchain

Jeanne Chaverot

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Semester Project

January 2018

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Jeffrey R. Allen
EPFL / DEDIS

Contents

1	Introduction	1
2	SkipChain Explorer	1
2.1	Cothority Framework	1
2.2	Choosing the right components	1
2.3	Talking to a SkipChain	1
2.4	Changing the Roster	2
2.5	Fetching the Blocks	3
2.6	Block Information	4
2.6.1	Backward and Forward links	5
2.6.2	Payload and Data	5
2.6.3	Verifiers	8
2.6.4	The Roster	8
2.6.5	Additional informations	9
2.7	Unit Testing	9
2.8	Issues	10
3	SkipChain Graph	10
3.1	Blocks and Links Representation	10
3.2	Usability	11
3.3	Issues	13
4	Measurements	14
4.1	Incentive	14
4.2	Results	14
5	Future Implementations	15
5.1	Look for a block	15
5.2	Add blocks	15
5.3	Optimizations	15
6	Personal SkipChain Explorer	15
6.1	Clone the project directory	15
6.2	Project setup	16
6.3	Setup your personal Roster	16
7	Conclusion	16

1 Introduction

SkipChains are cryptographically-traversable, offline and peer-to-peer-verifiable blockchain structures. A SkipChain is traversable in both directions, such that one party can efficiently prove the correctness of a transaction anywhere in time with respect to the other party's reference point on the blockchain, in a logarithmic number of steps, regardless of which party has a more up-to-date view of the blockchain [1]. Today, we have the public Cothority status dashboard at <http://status.dedis.ch/>, which gives no detail into what blocks and data are stored on the SkipChain. How do we inspect and verify the data that is stored in a given SkipChain? The goal of this project is to develop a platform allowing users to navigate through a SkipChain's blocks and their content.

2 SkipChain Explorer

2.1 Cothority Framework

The Cothority project (collective authority) is a decentralized cryptography Framework developed by the Decentralized and Distributed Systems lab (DEDIS) at EPFL. The Framework [2] will be used in the context of this project. In particular, the code in this repository allows the user to access the services of a cothority and/or run its own conode.

2.2 Choosing the right components

Before jumping into what Cothority has to offer, it is important to have an idea in mind of what we are working towards. This project uses Vue.js and Vuetify for the application rendering. First of all, several existing Blockchain Explorer tools have been analyzed and reviewed, such as etherscan.io and blockexplorer.com, to help choosing the best features to be implemented.

The first draft contains the home page, where the latest blocks from the given SkipChain would be displayed with some of the most interesting and general information about each of these blocks.

In order to recreate this draft, it was necessary to do some initial research on which Vuetify components to use and then on how to interact with SkipChains.

2.3 Talking to a SkipChain

As a first step, everything needs to run locally to make sure it works as expected. A local Roster is set up (i.e a group of Cothority servers) with

three running nodes:

```
1 $ ./run_conode.sh local 3 3
```

Once the local Roster is all set, it is possible to start generating SkipChains via the SkipChain Manager (scmgr) tool. Using the SkipChain Manager, the user can set up, modify and query SkipChains. The scmgr will be running on the local machine and it will communicate with one or more remote conodes. For it to work, the user needs the public.toml of a running cothority where he has the right to create a SkipChain or add new blocks. This command allows the user to generate interesting SkipChains (in this example, a SkipChain with base 3 and height 4 is created):

```
1 $ scmgr s c -b 3 -he 4
2 ~/go/src/github.com/dedis/cothority/conode/public.toml
```

Once the SkipChain is created, scmgr will print out its ID, and the user can start adding data into the chain:

```
1 $ scmgr s b add --data 'Hello' { skipchain id }
2 $ scmgr s b add --data 'World' { skipchain id }
```

2.4 Changing the Roster

As mentioned previously, the Roster is the group of Cothority servers which will sustain the SkipChains. A button on the upper right of the page allows the user to change the Roster he wishes to connect to via a simple text field. In this project, the default Roster is set to be the following:

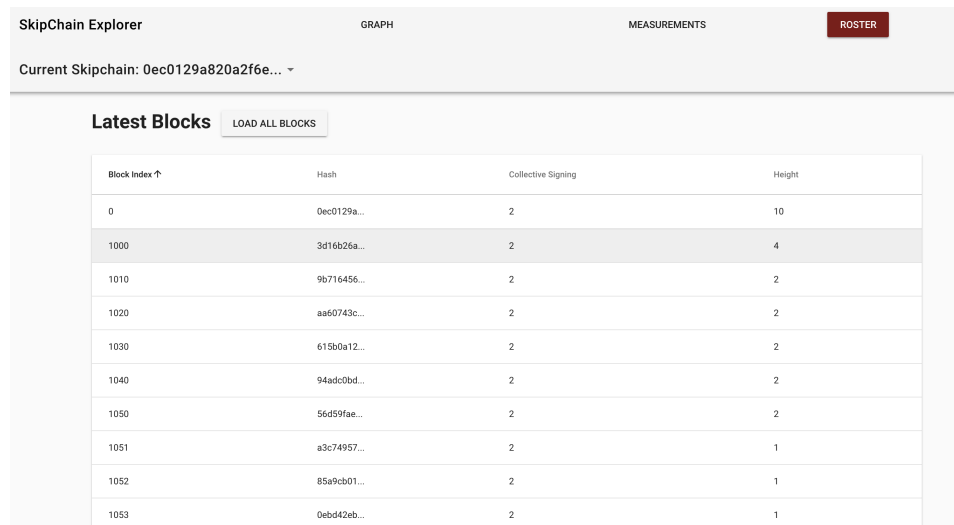
```
1 '
2 [[servers]]
3   Address = "tls://gasser.blue:7002"
4   Public  = "036bf316e1ea6e7e99e0bb713419d16
5             c0b6794bf9dc442cc4cf36c3f935e93cf"
6   Description = "EPFL Cothority-server"
7 '
```

which allows the user to have access to the DEDIS' SkipChains.

2.5 Fetching the Blocks

In this step, the <https://www.overleaf.com/project/5c238c1dddbc4b53474c059cgoal> is to interact with a SkipChain. However, as the web front-end is written in JavaScript using Vue.js and the back-end is written in Go, Protocol Buffers are needed. Protocol Buffer messages are a fast way to allow back-end/front-end communication [3]. To access a SkipChain's blocks the method `GetUpdateChain` is used. A first surprise was to see that it didn't seem to load all of the chain's blocks. After some research, it was realized that this method returns all the blocks from the highest layer of forward links. In other words, all the blocks from the fastest SkipChain's traversal.

Another option would have been to call `GetSingleBlockByIndex`. Although this method returns all the blocks, it is based on an iterative call, and would therefore kill the essence of the SkipChain: a logarithmic chain traversal. Only the "main blocks" from these highest links were then kept. In the home page, (status.dedis.ch/ng) the user is given the possibility to choose the SkipChain that he wishes to connect to. After selection, the latest blocks are displayed from the selected SkipChain, which are the blocks from the response of `GetUpdateChain`.



The screenshot shows the 'SkipChain Explorer' interface. At the top, there are navigation tabs: 'GRAPH', 'MEASUREMENTS', and 'ROSTER'. Below these, the current SkipChain is identified as '0ec0129a820a2f6e...'. The main section is titled 'Latest Blocks' and includes a 'LOAD ALL BLOCKS' button. A table displays the following data:

Block Index ↑	Hash	Collective Signing	Height
0	0ec0129a...	2	10
1000	3d16b26a...	2	4
1010	9b716456...	2	2
1020	aa60743c...	2	2
1030	615b0a12...	2	2
1040	94adc0bd...	2	2
1050	56d59fae...	2	2
1051	a3c74957...	2	1
1052	85a9cb01...	2	1
1053	0ebd42eb...	2	1

Figure 1: Homepage after SkipChain selection. Blocks and their main info are displayed

2.6 Block Information

The "Block Information" page is different for each block. It displays all the information considered interesting about the selected SkipBlock.

First of all, a method called `getBlockByHash` has been created, which, given a hash, returns the corresponding SkipBlock. This method will be particularly useful when working with Backward and Forward links.

Once the user has selected the desired SkipChain, he can navigate through its blocks, which is where it becomes interesting. What information is contained in a given block? How many blocks are reachable via the Forward/Backward links? The Block Information page gives the user the following:

← Block 1000, 3d16b26a997c4a509e4b6b6f3d4f63f108bbe58c2b3f2da83afda77dc819d5c →	
Backward links	▼
Forward links	▼
Payload	▼
Data	▼
Verifiers	▼
Roster	▼
Height	4
Max height	10
Base height	10

Figure 2: Information page for a selected block

2.6.1 Backward and Forward links

The Backward and Forward links of a block are displayed in function of their level. The higher the level the faster the SkipChain will be traversed, as it implies a bigger hop. The user can click on the link and he will be redirected to the correspondent block page. As Forward and Backward links are represented as hashes (and not SkipBlocks) of the block they point to, `getBlockByHash` had to be used.

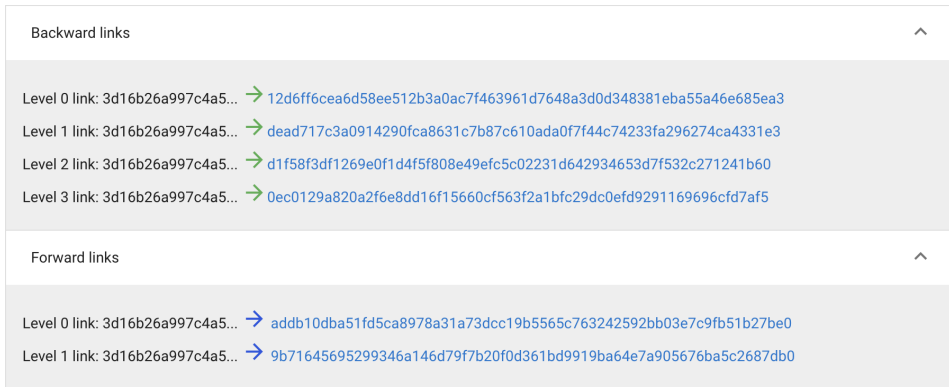


Figure 3: Forward and Backward Links of a given block

2.6.2 Payload and Data

The Payload and Data correspond to the Data Body and Data Header of a SkipBlock. In the context of this project, there were two types of SkipChains: ByzCoin and non-ByzCoin chains. A ByzCoin, by definition [4] is a Bitcoin-like crypto-currency enhanced with strong consistency, based on the principles of the well-studied Practical Byzantine Fault Tolerance (PBFT) [5] algorithm. First of all, in the case of non-Byzcoin SkipBlocks the Payload field of the block is null and the Data field is left as a hex dump.



Figure 4: Data Body (Payload) and Header (Data) of a non-ByzCoin Skip-Block

ByzCoin SkipBlocks are particularly interesting as they can hold multiple transactions per block. As a second step of the project, once everything was working, ByzCoin blocks have been decoded by recognizing whether there is a ByzCoin Verifier among the SkipChain's Verifiers set. In order to work with ByzCoin, it was necessary to stay on the byzgen_1810 branch, as it was stable. Decoding the Payload and Data of the SkipBlock becomes now more interesting.

ByzCoin Data

The following command allows us to inspect a block's Data (Data Header):

```
1  const headerLookup = protobuf.root.lookup('DataHeader')
2  const header = headerLookup.decode(this.block.data)
```

The Data Header field for ByzCoin blocks contains four elements: three hashes: client transaction hash, state change hash and the trieroot defined as the following:

- client transaction hash: a hash of all the client transactions listed in the Data Body (Payload) part of the SkipBlock
- state change hash: hash of the state change, which is either Create, Update or Remove
- Trieroot: hash of the Merkle trie root of Global State (whose value is determined by StateChange)

and one timestamp.

These values were left as they are, except for the timestamp which has been converted into readable format.

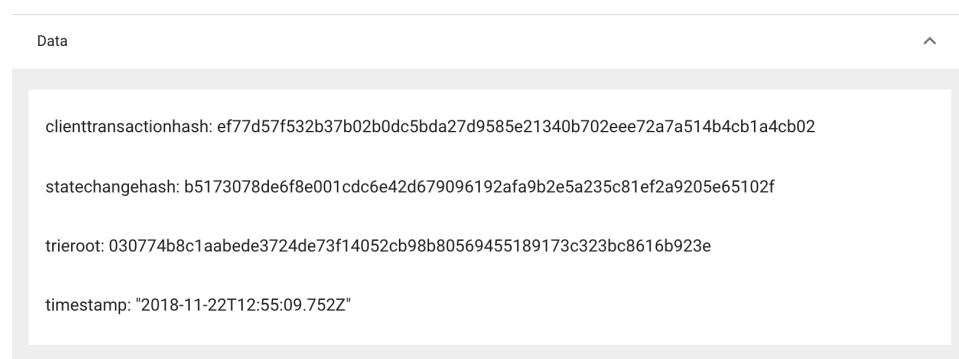


Figure 5: Data Header of a ByzCoin SkipBlock

ByzCoin Payload

As for the Data Body (Payload), we can inspect it with a similar command:

```
1 const bodyLookup = protobuf.root.lookup('DataBody')
2 const body = bodyLookup.decode(this.block.payload)
```

The payload is displayed as a list of transactions, which can either be accepted or rejected. We have two types of transactions: Spawn and Invoke (we haven't worked with Delete transactions, as there were none available). Spawn and Invoke transactions both contain a list of instructions and a signature. The signature for each transaction is represented as a tuple of a Signature (UUID) and a Signer. Spawn instructions are characterized by a contract id and an instance id (Figure 6). Invoke instructions are characterized by a command (Figure 7). All instructions then contain their arguments with name and value, which has been left under hexadecimal format.

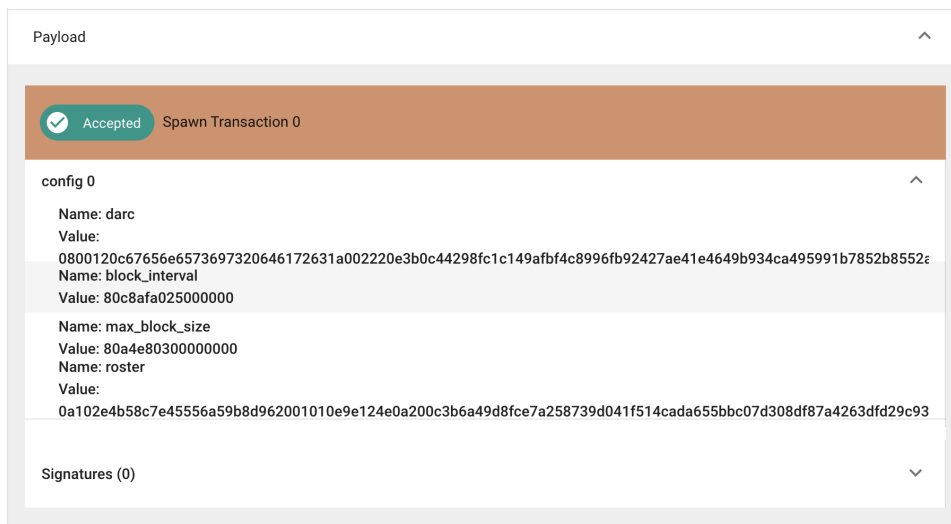


Figure 6: Payload of a ByzCoin SkipBlock as a Spawn Transaction with contract id "config" and instance id "0"

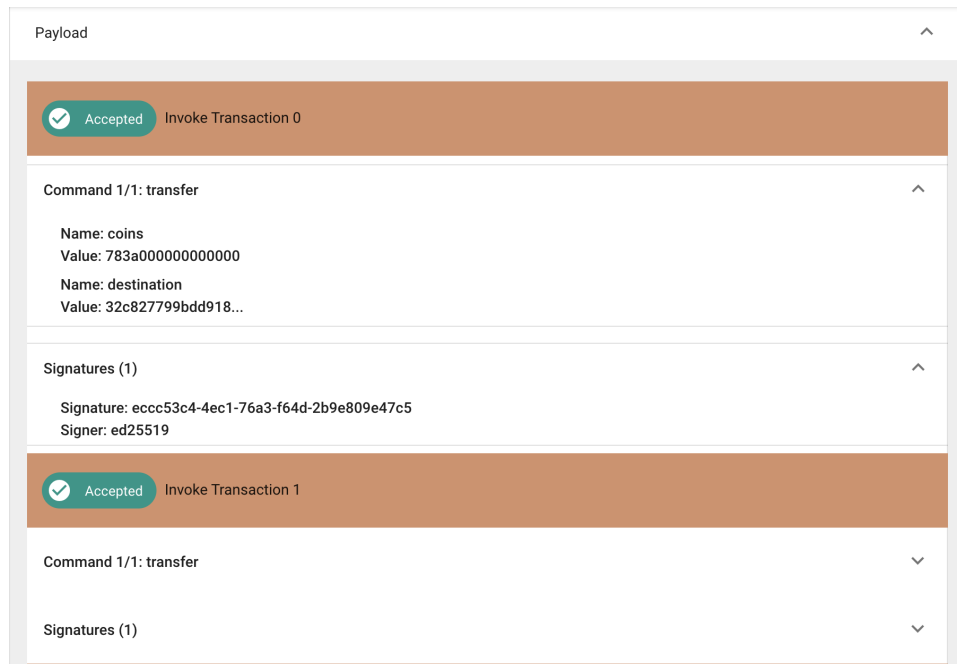


Figure 7: Payload of a ByzCoin SkipBlock as list of Invoke transactions with "transfer" commands

2.6.3 Verifiers

Verifiers tell the user whether this is a ByzCoin block or not. So far, the block information page displays two kinds of Verifiers: Base Verifier and ByzCoin Verifier. Each Verifier is displayed as a UUID.

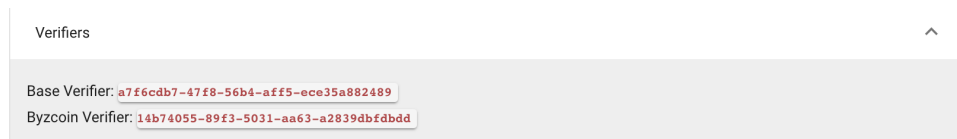


Figure 8: List of Verifiers for a ByzCoin SkipBlock

2.6.4 The Roster

In this part, we display the information relative to the Roster. A Roster is defined by its id and a set of conodes. Each conode has an id, an address, and a potential description. Each id is displayed as a UUID.



Figure 9: The Roster of a given block

2.6.5 Additional informations

Additional information such as height, max height and base height give details about the current SkipChain and SkipBlock. The bigger the base height, the longer the links between the blocks get. The bigger the maximum height, the longer the longest link gets and the more links a SkipBlock can have.

Height	4
Max height	10
Base height	10

Figure 10: Other informations about the given block

2.7 Unit Testing

Part of this project was to implement unit tests in order to verify that everything is displaying accordingly.

Although Vue.js suggests its users to use Karma, this project uses Mocha for unit testing as it is more straightforward and the most used library. Mocha has proven to be useful to cover the basic tests that were needed in this project. Two SkipBlocks that are connected to each other were saved in a 'blocks.js' file. Overall, it has been certified that the Block Information page displays every item correctly, for instance the block's hash, the UUIDs of the Roster and Verifiers. It was also checked that the hashes of the Forward and Backward links were displayed as expected in the Block Information page.

A test coverage has also been implemented. Once the tests passed correctly, Travis was added into the project. Travis is a great way to check whether all tests pass without forcing the user to run them himself. If the user wishes to test the code before pushing its changes, the following commands can be used:

```

1  $ yarn pretest --fix
2  $ yarn test
3  $ yarn coverage

```

At the end, creating tests was a great learning. Knowing that everything works as expected and leaving the code in a good state for the next programmer is rewarding and necessary.

2.8 Issues

One idea was, in the Forward/Backward links part of the block information page, to show next to each link to which block index it redirects to. This way, it would allow the user to know that with a click he can go from block 0 to block 1000, for instance. However, as Forward/Backward links are represented as hashes and not SkipBlocks, it would be necessary to fetch the SkipBlock (using `getBlockByHash`) to have access to its index, which would be time consuming. As a result, the index of the SkipBlocks are not displayed.

In this part, the reactivity of the components has major importance. The user must be able to navigate quickly between the blocks and the page design must be intuitive. Some bugs have been resolved, they resulted from a keying problem. Indeed, in Vue.js the pages update themselves based on a key given as input. If the programmer doesn't choose the right one it will result in a static page.

3 SkipChain Graph

The goal of this part is to create a visual identity to each SkipChain, allowing the user to see its format and interact with it. First of all, we looked for different libraries that could help reaching this goal. At the end, it has been decided to use D3, a JavaScript library that helps bringing data to life. Although there is a lot to take from D3, it was important to stick to the basics and work from there. A method called `getBlockByIndex` will be helpful in this part.

3.1 Blocks and Links Representation

To create the blocks and links, many instances have been used such as rectangle, line, polyline and polygon. The design has been inspired on the original paper [6]. For more usability, the graph has been changed from horizontally to vertically. This way, the user can scroll through the blocks more easily. As a first step, all the blocks that had been fetched were displayed, that is the blocks that have been returned by the `getUpdateChain` call. Indeed, as we remember, `getUpdateChain` only returns the blocks from the highest Forward link traversal (see Figure 11 (a)). In the graph, each block's size is represented as a multiple of its height and the blocks are ordered in function of their index.

Once that was in place, the rest of the blocks (the un-fetched ones, non returned by the `getUpdateChain` call) were displayed. The user has the possibility to fetch any of these blocks by clicking on it. These unloaded blocks are represented as grey squares. Unloaded blocks from higher layers (i.e they have forward links of level 1 or higher) are represented in light-pink. When the user clicks on a square, it fetches the corresponding block via a `getBlockByIndex` call. The user can also have access to the information of a certain block by double-clicking it: he will be redirected to the information page of the block in question. We only display links from blocks that have been fetched.

3.2 Usability

A user might want to have an easy access to the latest block. As the graph visualization starts with the Genesis block, a button that allows the user to go directly to the bottom of the page was added. This way, one can directly access the end of the SkipChain (i.e the latest SkipBlock). The opposite has also been implemented, once at the bottom of the page, the user can go back to the top with a simple click.

One might also want to load the whole SkipChain, i.e all the grey squares, or at least its majority. For this purpose, two buttons were added:

- Load Higher Traversal Blocks (see Figure 11 (b))
 - This button will fetch all the blocks from the non-zero link layers (represented in light-pink). That is, the only resulting unfetched blocks are the ones from the lowest layer.
 - This feature allows the user to have a good visualization of the SkipChain without having to load it entirely.
- Load All Blocks (see Figure 11 (c))
 - This button will fetch every single block from the corresponding SkipChain.
 - A pop-up will appear to confirm the action. Indeed, if the Roster isn't potent enough, the web-browser might crash.

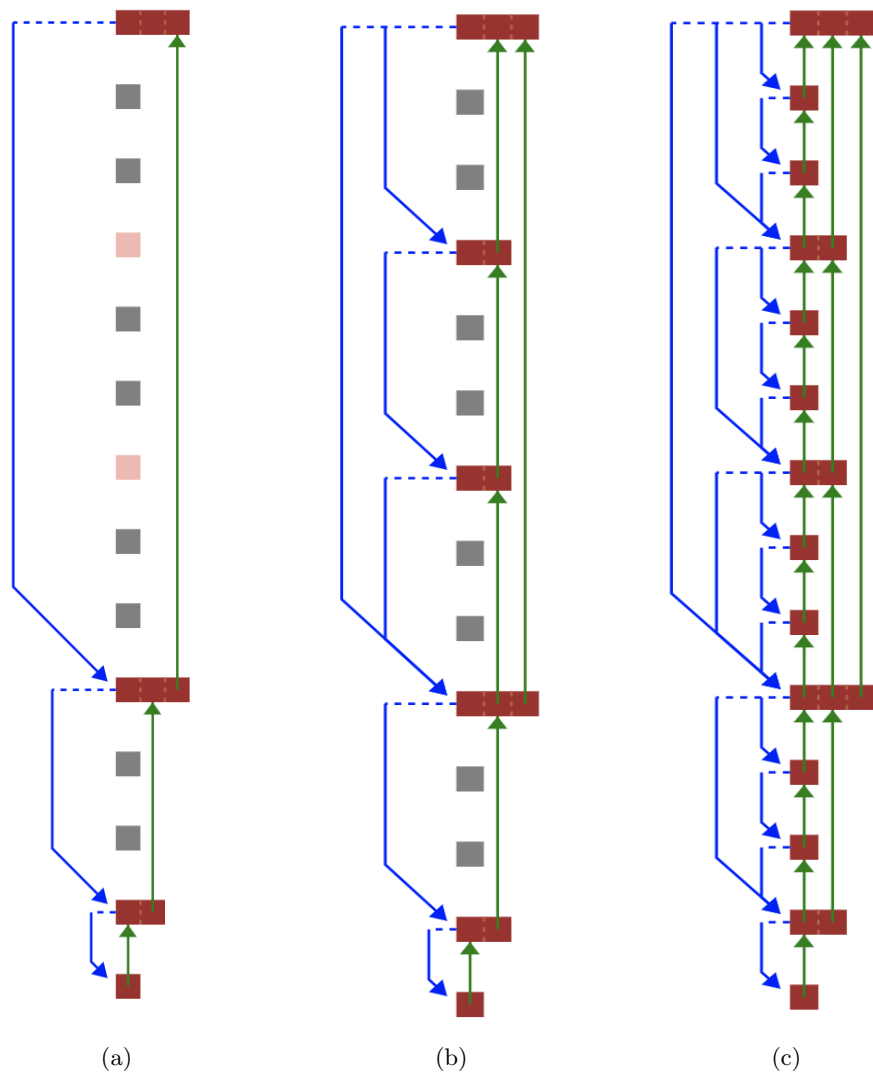


Figure 11: The current SkipChain's graph after (a) loading the Graph page (b) fetching most SkipBlocks from highest link layers (c) fetching the whole Chain.

3.3 Issues

One issue has come up when creating all blocks. Indeed, once `getUpdateChain` returned some blocks, it was necessary to artificially create the rest of the blocks (the unloaded ones) of the `SkipChain` in order to be able to display them as well on the graph. Meaning that there is an array called `allBlocks` containing either blocks (that have been fetched) either simple objects with the following definition:

```
1 { loaded: false, index: i, height: 1 }
```

The height is preset to 1 so they can be displayed as squares, but once the `SkipBlock` is fetched its real height is assigned to it. At this moment, an indexing issue has been discovered: instead of showing a block's index `i`, it showed `2*i`. After some research and re-reading of the code, to conclusion that has been made was that the problem came from the Protocol Buffering on the lab' side. Once this issue was taken care of, indexes were displayed as expected.

Some work has been done so that the `SkipChain's` Graph displays correctly on mobile platforms, by adapting its dimensions.

4 Measurements

4.1 Incentive

So far, a visualization tool has been created for the SkipChains from a given Roster and their content. The SkipChain technology promises a faster chain traversal in logarithmic time in comparison to other BlockChain technologies. Has this development been able to keep that feature? Are SkipChains still profitable and time-saving? To verify these hypothesis, a series of measurements has been conducted. The following is being computed:

- First of all, how much time does it take to traverse every Forward link layer. That is, how much time it takes to go from the Genesis block to the latest block on the given layer of Forward link. This is computed based on the `getBlockByHash` method.
- Then, it is interesting to see how long it takes to fetch every single block from the chain index by index, using `getBlockByIndex`.
- Finally, we measure how much time it takes to traverse the whole chain, how long it takes to get to the latest block starting at the Genesis block. To do so, we stay on the highest Forward link layer at all times. The search is done using `GetBlockByHash`.

4.2 Results

We will observe the results in the largest SkipChain we have so far, which has 1'058 blocks to this date, and four different layers of Forward Links. It is relevant to note that this SkipChain has base 10 and height 10.

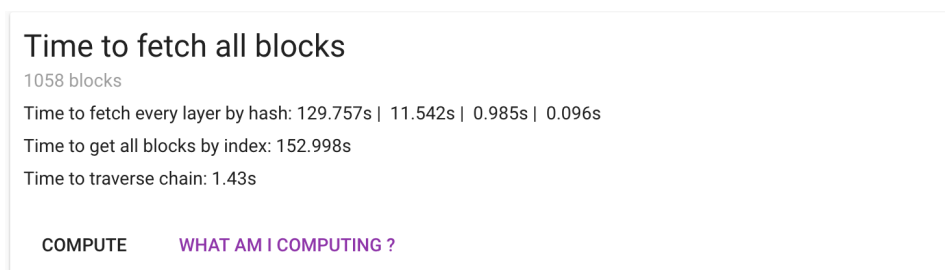


Figure 12: Measurements for a 1'058 SkipBlocks SkipChain

As we can see it takes up to 2 minutes (129.757 s) to traverse the lowest layer and 0.096 seconds to traverse the highest one with the `getBlockByHash` call. It also takes up to 2 minutes (152.998 s) to fetch all blocks using `getBlockByIndex`. Finally, it takes a total of 1.43 seconds to traverse the

whole chain. It can indeed be concluded that using higher level Forward Links brings efficiency to the chain's traversal. More precisely, using the highest Forward links rather than sticking to the lowest level Forward link is more than ninety times more efficient ($\frac{129.757}{1.43} = 90.73$). Of course, these numbers are not static and will change from one measurement to the other.

5 Future Implementations

In this part, we will make a list of suggestions to improve this project.

5.1 Look for a block

A first feature could be to give the user the possibility to find a block based on its hash. This would allow any user to verify a certain information is indeed stored on the SkipChain. This feature could have the format of a search bar to which we could even add different filters.

5.2 Add blocks

It could be interesting to give the user the possibility to add informations, transactions, on the current SkipChain directly from this interface.

5.3 Optimizations

Finally, we could agree to optimize some of the already implemented functionalities. For instance, we could change the Roster button. If many Rosters exist, we could show a list of popular Rosters to which a user would choose to connect to. When we have really large SkipChains, the Graph part might be a little blurry and confusing. It could be interesting to have a mini-map that allows the user to have an overview of the SkipChain's format.

6 Personal SkipChain Explorer

How to run this project to visualize your own SkipChains.

6.1 Clone the project directory

First of all, clone the project in the desired location

```
1 $ git clone https://github.com/dedis/student_18_explorer
```

6.2 Project setup

In the command line of the project directory, run the following commands:

```
1 $ yarn install
2 $ yarn run serve
3 $ yarn run build
```

And open the project in your navigator (default <http://localhost:8080>)

6.3 Setup your personal Roster

By default, you'll be connected to DEDIS' Roster and have access to its SkipChains. If you want to interact with your own SkipChains which are running on your personal Roster, click on the Roster button on the top-right of the page, and paste the data from your .toml file into the text component. The page should directly load all your SkipChains.

7 Conclusion

SkipChains are strongly connected components that allow a fast content retrieval. Although this technology is still under heavy development, we were able to see and measure its efficiency. Far more than being able to store data into SkipChains, we are now able to inspect its content. Even better, we can now also interact with ByzCoin SkipChains and verify if a transaction has indeed been stored with an interactive and visual application. An interesting next step would be to fully optimize this implementation for mobile users and keep it up to date with already existing projects, as e-voting for instance. On a personal note, this project has allowed me to discover many aspects of software development: From working with Vue.js, Vuetify and D3 in the front-end to interacting with the Cothority Framework and creating unit tests on the back-end, it has definitely been enriching. As a last comment I would like to thank all the engineers that have to this day worked on the Cothority Framework. A special thank you to Linus Gasser who's kept his conodes available so I could work with his SkipChains, and to my mentor Jeff Allen who's fed my journey with honest opinions and good advices.

References

- [1] Bryan Ford: How do you know it's on the blockchain? With a SkipChain (2017)
- [2] DEDIS: Scalable collective authority original documentation and project
- [3] @dedis/cothority: Cothority client library in Javascript
- [4] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford: Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing
- [5] CASTRO, M., AND LISKOV, B. Practical Byzantine Fault Tolerance. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Feb. 1999).
- [6] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, Justin Cappos and Bryan Ford: CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds
- [7] Jeanne Chaverot: SkipChain Explorer: original project