CDC Project AS22 - Basic Blockchain in Erlang

Group 3: Elleby, Gabriel, Odermatt

December 22, 2022

Contents

1 Introduction & Problem Statement

As part of the 5th-semester bachelor's course at the Université de Fribourg, *Concurrent and Distributed Computing*, an introduction to Erlang, Elixir, and the fundamentals of concurrent and distributed computing is given. Also part of this course is a project and this report.

The main goals of this project are:

- Deepen your understanding of the fundamentals learned in the course
- Analyze the characteristics & constraints of a distributed computing problem and come up with a solution
- Implement a distributed algorithm in Erlang or Elixir
- Prove your autonomy and initiative in the realization of such a project as well as in the discussion of the results

Regarding the project topic, we were able to choose from a given list or come up with our own. The topic we have decided on tackling is a simple Blockchain.

Since this is a project as part of a one-semester course, the focus is laid on the basic features of a blockchain. Henceforth, existing blockchain implementations and use cases will not be addressed extensively.

2 State-of-The-Art

This section deals with the basic concepts of a blockchain. It briefly explains what a blockchain is and its basic concept. Additionally, some well-known Implementations are named.

2.1 A Blockchain

As the name implies, a blockchain is a sequence of blocks linked together to form a *chain*. A block needs to contain at least the hash of the previous block, also called *parent* block and the timestamp of when the current block was created. The parent hash links the current block to the previous one. This ensures that no block can be inserted between two already present blocks and no data of a specific block can be changed. If a malicious actor tries to insert a faulty block, it is recognized by checking the parent hash of the following block. If these hashes do not match, the previous block has been inserted maliciously and is not accepted. If data is changed, the hash also changed and the child block's parent hash does not match up.

The real power of a blockchain comes with it being distributed. In a local blockchain, a malicious actor can still rewrite all of the local blockchain's contents to match up. On a distributed network where every node of the system has the same copy of the blockchain the amount of effort required to tamper with it, i.e. changing every node in the network, makes this pretty much impossible.

2.2 The Basic Elements of a Blockchain

Besides the basic structure of a blockchain, i.e. how a block is used in the blockchain and what data it must hold, the following aspects are also necessary:

- **Shared record:** Every node in the network always has an up-to-date copy of the entire blockchain.
- **Consensus:** A new block added to the blockchain has to be approved by the system. This can be done in different ways. There exist some well-known concepts. One is **proof-of-stake**, where the ability to create a new block depends on the amount of the *holdings* a given node has in the network; often used in cryptocurrencies. Another one is **proof-of-work**, where the creation of a new block depends on the computational power a node has invested into securing the network. A third approach is a **majority vote** algorithm, where the majority of participants vote on whether or not a new block is added to the blockchain. This is only a selection of different concepts which

might be adapted for a specific system. The actual choice depends heavily on the desired use case of a blockchain.

- **Content:** This part deals with the specific requirements a participant has to adhere to in order to add a new record to the network. In Bitcoin, for example, a block not only needs to have a special hash value, but also its content needs to be valid. This means it can only consist of legal transactions. A block with arbitrary data, for example, a *JPEG* image, will get rejected by the network.[1]

Depending on the use case, some more requirements can be considered. But for the basic implementation of a blockchain, these are sufficient.[2][3]

2.3 Existing Programs Using A Blockchain

There already exist numerous different blockchains[4], the most famous among them being Bitcoin and Ethereum. Additionally, blockchain implementations are already done in different programming languages, Erlang being one of them[5].

3 Presentation of the Solution

In this section, the main features of a blockchain and some additional ones needed to help run the program, are discussed. Additionally, some notes regarding certain aspects of the final implementation are given.

3.1 Blockchain Functionalities

3.1.1 create_block()

The most important functionality of a blockchain is being able to create blocks. For this implementation, each block contains three parameters: the parent's hash, specified data, and the timestamp of when it was created with *erlang:timestamp().* This function is called *create_ block().* The *Blockchain* is passed as a parameter from which the different elements needed as arguments for the new block are taken. It creates a parent by using the last block and the specified data, which can be of any type and length - no constraints are set for this basic implementation.

3.1.2 genesis()

Every blockchain needs a starting point - often called *Genesis* block. This is the first block in the chain and thus does not have a parent block.[6] In our basic blockchain the genesis block is hardcoded in a special function called *genesis().* This function does not take any parameters and returns a genesis block. The parameter for the parent hash is set to the atom *genesis*, the data to `{"genesis",0,0,0}`, and the time to *0,0,0.*

3.1.3 Consensus Algorithm - Proof-of-Work

Regarding the consensus algorithm, we did opt for a proof of work algorithm. This decision was mainly motivated due to the relative simplicity of implementing it. Erlang already provides a *crypto* module with a *hash()* function. We choose *md5* for this because for this proof-of-concept it is not necessary to make it tamper-proof with a stronger algorithm. The proof-of-work is implemented in a way where the hash calculated is checked for a zero in the front *(leading zero)* which is proof of the work done for the network. The parameters given to the hashing function *hash_ block()* is a new block containing the next to include *content* from the *Mempool* (see *Mempool 3.1.4*), *self()*, i.e., the current process, the length of the blockchain, and a *Nonce* value, which represents the number of tries this process made to create a block. This *Nonce* and the timestamp serve as the changing values in the input for the hash function. Otherwise, the same hash would be created over and over again. Upon successful generation of a hash with a leading zero or the reception of a valid block, the *Nonce* is reset to zero.

3.1.4 Mempool

The *Mempool* is a list of data to be added to the blockchain of which every process has a copy. Data can be added to the list by sending the process a message in the form `{mempool, Msg}`. When a process creates a block, it includes the first element of that list in the block and removes it from the list. When a process receives a new block, it checks the data field and removes the first element in its local *Mempool* that matches the message in the block. With the function *send_ to_ mempool(Group,Msg)*, data can be sent to every blockchain process which will then be included in the next block created. Since the messages of the *Mempool* and the blocks are not sent using atomic multicast, it is possible that a message gets included into the blockchain more than one time.

3.1.5 main()

The *main()* function handles message reception. Two versions of this function exist. This *initializer* function, which only takes *UX* & *Blockchain* as arguments and it is used during the initialization phase of the program, is called when a process is spawned so that the process waits until it receives the *init* message. Also, that way, no empty arguments have to be given when the process is spawned.

The other *main()* function coordinates everything within a member of the network. There are multiple receives for different events. The first receive - `{update, PID, Block}` - is used if another network member broadcasted a new block to the blockchain. Since the *Group* variable known to *main()* contains every process, if one broadcasts the new block to the *Group*, it is also sent to itself again. This is checked with the `PID == self()`. Otherwise, a check is made whether or not the block is valid, i.e., the new block's parent hash lines up with the last block's hash from the blockchain, and the new block's hash has a leading zero. If those two checks succeed, the block is accepted to the blockchain. Also, if a message from the mempool was included, it is deleted from the process's mempool with the function *remove_from(Msg, Mempool)*.

Other messages the main function can receive are `{mempool, Msg}`, used to add data to the process's mempool, and `{getBlkCh, Sender}` which sends back the blockchain to the *Sender*. `{getGroup, Sender}` and `{getMempool, Sender}` are also used by the user interface and return the *Group* and *Mempool* variable respectively, see *User Interface 3.2.5*. Upon reception of `{terminate}`, the process terminates.

Lastly, if `after 50 + R` milliseconds, where `R = rand:uniform(50)`, no new message was received by the process, an attempt to create a new block will be made.

3.2 Helper Functions

3.2.1 start()

To start the program, the *start()* function has to be used. There exist multiple versions of this function depending on how many arguments are entered. The most basic is the one where no arguments are entered. Here, info about the command that is needed to be able to run the program and what arguments are expected for the function, are printed. The first argument of the function defines whether one wants to run the current nodes program in user interface mode or not. The second argument to be entered is the name and hostname of the other Erlang shell with which to establish a network. Finally, the third one is the number of mining processes this shell should create. In this implementation, there can only be two shells connected, but the programmer can choose n-many mining processes to create inside of these two nodes. This is elaborated in more detail in the section *Discussion of Results 5.* More information on how to run the program can be found in the *Appendix - How to Run the Code A.1.*

3.2.2 deploy()

After the genesis block is created, the *start* function calls *deploy()*. It spawns the requested amount of processes by using the in subsection *3.1.5* described *initializer* main function and returns a *Group* with the process identifiers.

3.2.3 pingNode(), exchGroup(), sendGroup() & recvGroup()

Since the implementation does not have a way of dynamically adding other nodes to the network, their connection has to be established before running the program. Moreover, the hostnames used have to be known in advance and can not be determined automatically by the program. Therefore, running the implementation in Erlang can be tricky. Each node that is launched must know the hostname of the other node. To validate the connection, the other hostname is handed over as an argument to the function *start()*. Here it is pinged until it received a response back in the form of a *pong* that shows that the nodes are connected indeed.

After the response, the two previously created process groups are exchanged between the two nodes by utilizing *exchGroup()*. To achieve this, two new processes are spawned in each node. The first with *sendGroup()*, where after a delay of three seconds, a message is sent to the receiving process of the other node. This message contains the current nodes *Group*. The second, utilizing *recvGroup()*, is created and registered as *recvG* for both nodes. The delay is necessary to give the processes time to be spawned. Otherwise, the sending fails. Since the name of the receiving process is known to the other node because they have the same name, it is possible to send a message to the *recvG* process from the other Node; the node is denoted by *OtherNode*. The *Group* from the sending node is then merged with the receiving node's *Group*. The ordering of the different processes inside the *Group* variable is undefined - it does not matter if upon a block's creation one process is called before another one. The new *Group* is then sent to the beforehand mentioned *initializer main()* function in the form `{init, Group}`, see *main() 3.1.5*.

3.2.4 print_blockchain & print()

To display the data, two different print functions have been implemented: *print_blockchain()* prints a given blockchain in a readable way. *print()* is a general-purpose print function with the possibility to print or not print specified data depending on whether or not the node is in user interface mode. This option is denoted by the first argument - *true* or *false*. As the second and third arguments for *print()*, the *String* to print and its corresponding *Arguments* are added. The first and last arguments are optional; *print()* can be used in the form `print("Any String")` or `print("~s", [String])`. The latter example represents a direct mapping to *io:format()*.

3.2.5 The User Interface

The User Interface provides a way for the user to directly interact with the blockchain. It is activated by setting the *cli* value in the running command to true. When it is enabled on a node upon starting the program, eight options can be chosen. With the first one, it is possible to specify data in the form of text, which gets included with the next block using the mempool functionality. The second option prints the entire blockchain utilizing *print_blockchain()*. By selecting the third option, the user can print the blockchain from a specific process denoted by its *Process ID*. The *Process ID*s to choose from are listed automatically. The fourth option prints the contents of all the processes' mempools. Option five lists all processes currently running in the network. Option six writes the time difference in seconds for when the blocks are created and the number of *Nonces* necessary to do so into a *.csv* file by utilizing *recordTime()* and *writeTime()*. Those two functions filter out the Genesis block. Since Genesis is initialized with a time of `{0,0,0}`, it does not make sense to calculate the time difference between it and the first block created. The function instead takes the first new block created by the blockchain network and starts calculating the time difference with the timestamp added upon the block's creation onwards. In option seven, a predefined faulty block is sent to the blockchain processes in an update message. Those will then print in the terminal that it is invalid. And the eighth option terminates the entire blockchain.

The user interface's process is created when the starting process has finished spawning the specified amount of blockchain processes. If specified, it becomes the user interface process. If the user interface is not selected, this starting process will terminate after about one second.

Regarding termination, two different functions exist: *terminate()* and *terminateNetwork()*. the first is used by the user interface process; one process is sent the signal to broadcast the termination signal to every other process of the blockchain network. The second function is used by the blockchain processes to broadcast the termination signal. To ensure every process receives the message, every process receiving the termination signal also broadcasts this message to the network.

3.3 Additional Remarks regarding the Implementation

3.3.1 The *Group* Variable

A remark regarding the *Group* variable: the starting process as well as every blockchain process have this variable. However, those two variables slightly vary in their contents. The *Group* variable the starting process uses for the user interface is of the form `{Integer,PID}`, whereas after the group exchange - the variable used for the main function - is of the form `{Integer, PID, Node}`. This difference is the reason why the user interface sometimes has to get the *Group* variable from a process it knows in the network first.

3.3.2 User Input Check

Some user input checks are implemented. In the options selection of the user interface, only numbers from the possible options are allowed - everything else will not be accepted. In option one, only newline characters '`\n`' and empty input *""* are not permitted. Since data for the blockchain can be entered as text, most should be allowed. With option three, however, the potential to input faulty data, which makes *beam* crash, is very high. The reason for this is that to convert the entered PID from a *String* to something Erlang can handle, *list_to_pid()* is used. This function crashes when anything else other than a *String* in the format *<X.X.X>* is used, where '<' and '>' have to be included and the '*X*' denotes a number, e.g., *<0.89.0>*. For now, there is no input check to mitigate this issue. Regarding *optionFour*, *optionFive*, and *optionSeven*, there is no need for further input checks because them not requiring any extra arguments from the user. Regarding *optionSix*, when the *Time.csv* is used by another program, *beam* crashes upon selecting this option because it cannot write to the file.

3.3.3 Color Formatting in Terminal

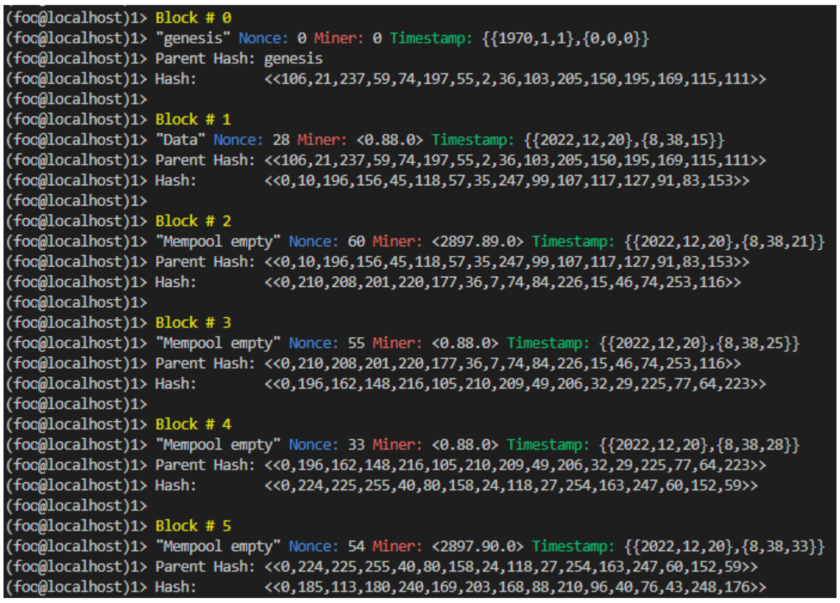
Upon running the program, the user can see text printed on the terminal formatted with colors. This is handled by the *color* module present in the code directory. The source can be found on GitHub [7].

4 Validation of Implementation

The implementation turned out well and runs successfully on two terminals. The two Erlang shells can spawn n-many processes. Depending on the Boolean value used in the running command, the shell either shows the inner workings of the backend or a user interface consisting of options to interact with the network. This video shows one shell just showing the backend and another shell being set as the User Interface: *Video*. Here you can see the different elements of the blockchain in action, as explained in section 3. The simple functions available in the User Interface are sufficient for the desired result. However, some might crash the program if the wrong input is given. This is especially the case when using the third option. More details about those issues can be found in *User Input Check 3.3.2*. When the correct commands are entered, the manipulation shows up on the terminal which prints the blockchain. This indicated the validity of the implementation.

When the two Erlang shells spawn a large number of processes, a concurrency issue occurs, observable in this *Video*. This issue occurs as the frequency of blocks being mined is increased and therefore also the probability that this happens simultaneously. That is also the reason why the printed blockchain shows that multiple blocks have the same parent hash, as processes are passing the newly mined block at the same time and therefore were not yet notified of the other block's creation. To deal with these forks, one could implement a tree structure where the newly mined blocks get added to a branch and the longest branch becomes the blockchain. More explanation can be found in *Discussion of Results*, see section 5.

Regarding the proof-of-work, the number of zeroes preceding the hash of a block decides whether or not the newly created block is added. The difficulty is hard-coded in this implementation. When the implementation is run, one can see that multiple nonces are tested before finding a block that validates the requirements of the proof-of-work, as seen in blue in Figure1. In the same screenshot, the timestamps of when the blocks were mined can be seen in green. The time difference between blocks signifies that many attempts are needed to find a block that has a hash with a zero in front. In Figure 1, the hash of each block can be compared manually to the parent hash field of the next block to make sure that the blockchain is intact. This is proof of the program working as intended.



```
(foc@localhost)1> Block # 0
(foc@localhost)1> "genesis" Nonce: 0 Miner: 0 Timestamp: {{1970,1,1},{0,0,0}}
(foc@localhost)1> Parent Hash: genesis
(foc@localhost)1> Hash:        <<106,21,237,59,74,197,55,2,36,103,205,150,195,169,115,111>>
(foc@localhost)1>
(foc@localhost)1> Block # 1
(foc@localhost)1> "Data" Nonce: 28 Miner: <0.88.0> Timestamp: {{2022,12,20},{8,38,15}}
(foc@localhost)1> Parent Hash: <<106,21,237,59,74,197,55,2,36,103,205,150,195,169,115,111>>
(foc@localhost)1> Hash:        <<0,10,196,156,45,118,57,35,247,99,107,117,127,91,83,153>>
(foc@localhost)1>
(foc@localhost)1> Block # 2
(foc@localhost)1> "Mempool empty" Nonce: 60 Miner: <2897.89.0> Timestamp: {{2022,12,20},{8,38,21}}
(foc@localhost)1> Parent Hash: <<0,10,196,156,45,118,57,35,247,99,107,117,127,91,83,153>>
(foc@localhost)1> Hash:        <<0,210,208,201,220,177,36,7,74,84,226,15,46,74,253,116>>
(foc@localhost)1>
(foc@localhost)1> Block # 3
(foc@localhost)1> "Mempool empty" Nonce: 55 Miner: <0.88.0> Timestamp: {{2022,12,20},{8,38,25}}
(foc@localhost)1> Parent Hash: <<0,210,208,201,220,177,36,7,74,84,226,15,46,74,253,116>>
(foc@localhost)1> Hash:        <<0,196,162,148,216,105,210,209,49,206,32,29,225,77,64,223>>
(foc@localhost)1>
(foc@localhost)1> Block # 4
(foc@localhost)1> "Mempool empty" Nonce: 33 Miner: <0.88.0> Timestamp: {{2022,12,20},{8,38,28}}
(foc@localhost)1> Parent Hash: <<0,196,162,148,216,105,210,209,49,206,32,29,225,77,64,223>>
(foc@localhost)1> Hash:        <<0,224,225,255,40,80,158,24,118,27,254,163,247,60,152,59>>
(foc@localhost)1>
(foc@localhost)1> Block # 5
(foc@localhost)1> "Mempool empty" Nonce: 54 Miner: <2897.90.0> Timestamp: {{2022,12,20},{8,38,33}}
(foc@localhost)1> Parent Hash: <<0,224,225,255,40,80,158,24,118,27,254,163,247,60,152,59>>
(foc@localhost)1> Hash:        <<0,185,113,180,240,169,203,168,88,210,96,40,76,43,248,176>>
```

Figure 1: Screenshot of the terminal with the blockchain printed

5 Discussion of Result

The implementation demonstrates the basic concepts of a blockchain and it works well in highlighting all its vital parts. Since the project only has a limited scope, this is not a final implementation of a blockchain in Erlang. It rather proves the possibility to create such a network using this programming language.

There are still issues in the implementation which can be improved:

- **Tree Structure:** In a large network, depending on the difficulty of the proof of work algorithm and the number or the speed of processes creating blocks, processes might create new blocks 'simultaneously' frequently - a process might create a new block and update the network accordingly, however, another process did also find a new block when an update message is pending in its queue. The second process still broadcasts the update. This then results in a fork of the blockchain, i.e., two processes have a new valid block added to their blockchain even though those two blocks are not the same. The processes then continue to run and add new blocks to their blockchains. The first process's last block is not the parent block of the block sent by the update message from the second process. Thus, it cannot be added and is deemed invalid by any third process. In the implementation, this is not yet handled, but it can be by implementing a tree structure to store blocks instead of just using a list. Then, multiple branches or forks of the blockchain can be kept in memory and the longest of them is regarded as the valid chain. We did try to implement it but only succeeded with a binary tree but this was not sufficient, as we need a tree structure with more than two possible branches.

- **Dynamic adding of participants in the network:** In this implementation the number of different nodes and processes cannot change upon initialization. In a blockchain network, participants can usually be added dynamically while the network is running.

- **Genesis creation:** The *Genesis* block is created in every node. Since the function responsible creates the block with the fields for the *parent hash*, *data*, and *timestamp* fixed, having each node do this is not an issue on its own. However, this is not the best way to implement it. A better way is to only let one node create a genesis block and distribute it to all other nodes.

- **Mempool:** The *Mempool* could be improved to provide more reliability. This can be achieved either by using atomic multicast for sending messages between participants or by adding a more sophisticated method of keeping track of what needs to be removed from the *Mempool*.

- **User Interface:** Besides adding more useful functionalities, faulty input might still crash the user interface's node and the corresponding blockchain processes. The main example is the third option, when no input in the form *<X.X.X>* is entered, the *list_to_pid()* function crashes. This might also be the case for other functions, however, besides the third option, this has not yet occurred to us during testing. As mentioned, some input check is already implemented for the first and second option.

  Additionally, the user interface code can be improved by making it get the updated group variable from the network once and storing it instead of having to request *Group* from a member every time the UI needs to send something to the entire blockchain.

- **Write the blockchain to a file:** The size of a blockchain greatly depends on how much data is included, which is different for every use case. For now, the blockchain is only stored in memory. This is not an optimal solution for a proper blockchain network as the blockchain can get very big - Bitcoin's blockchain has a size greater than 400GB [8]. Since the entire blockchain does not need to be accessible fastly, having it stored only in memory is not necessary. Storing hundreds of GB in memory also poses an enormous demand for costly hardware to participants. Also, in the case of a power outage and subsequent loss of the blockchain must be avoided.

- **Docker on windows terminal:** The implementation was also tested using Docker Desktop on windows. Two containers were created via separate terminals (one for each Erlang shell). The code did compile successfully and could be run on the two terminals, one showing the current blockchain and one showing the user interface. The issue was, that when something was entered in the terminal, the input was not being registered and therefore the blockchain could not be manipulated interactively by the user interface. The problem seems to be related to the *io:read()* function in combination with Linux terminals.

In conclusion, we succeeded in implementing a basic blockchain in Erlang and were able to verify that it works correctly. We also provide a command line interface as a frontend to interact with the blockchain, namely by adding custom content to a block, printing the blockchains stored on different processes, and trying to tamper with it.

Furthermore, we already identified possible improvements that could be implemented in the future and are confident, that our skills in working with distributed systems improved in the course of working on this project.

Bibliography

[1] *Blockchain.* URL: `https://developer.bitcoin.org/devguide/block_chain.html`. (last visited: 09.12.2022).

[2] Manav Gupta. *Blockchain for Dummies, 2nd IBM Limited Edition.* John Wiley Sons, Inc., 2018. ISBN: 978-1-119-54593-4 (pbk); ISBN: 978-1-119-54601-6 (ebk).

[3] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.*

[4] Tien Tuan Anh Dinh Rui Liu et al. *Untangling Blockchain: A Data Processing View of Blockchain Systems.* URL: `https://arxiv.org/pdf/1708.05665.pdf`. (accessed: 21.12.2022).

[5] Rahul Garg Andrew Thompson et al. *BEAMCoin.* URL: `https://github.com/novalabsxyz/BEAMCoin`. (last commit: 09.04.2018).

[6] Carla Tardi. *Genesis Block: Bitcoin Definition, Mysteries, Secret Message.* URL: `https://www.investopedia.com/terms/g/genesis-block.asp`. (accessed: 14.12.2022).

[7] Duncan McGreggor Julian Duque Evgeni Kolev. *erlang-color.* URL: `https://github.com/julianduque/erlang-color`. (last commit: 04.11.2016).

[8] Raynor de Best. *Size of the Bitcoin blockchain from January 2009 to July 11, 2022.* URL: `https://www.statista.com/statistics/647523/worldwide-bitcoin-blockchain-size/`. (accessed: 19.12.2022).

A Appendix

A.1 How to Run

A.1.1 The Repository - Code

The repository of this project can be found on GitHub.

A.1.2 Requirements

You need the files *project.erl*, *color.erl* and *color.hrl* and compile the erl files using Erlang. For the program to run, you need to start it in two terminals. The program does not run until it connects to the second instance.

A.1.3 Commands

First, compile the project.erl and color.erl file in an Erlang shell with c(project). & c(color).
Then use the following command to run the program in the terminal:

```
   erl -sname #{name@host} -setcookie #{secret} -s project start #{cli}
#{otherName@otherHost} #{NumberofProcesses}
```

- #{name@host} is the sname and hostname of the local Erlang VM
- #{secret} is the shared secret
- #{cli} true for running in interactive (CLI) mode, false for seeing the automated 'back-end'
- #{otherName@otherHost} is the sname and hostname of the remote (other) Erlang node
- #{NumberofProcesses} is the # of processes that will mine new blocks, more = the blockchain grows faster and the likelihood of forks increases

Examples:

```
erl -sname foo@localhost -setcookie test -s project start true bar@localhost 3
erl -sname bar@localhost -setcookie test -s project start false foo@localhost 60
```

A.1.4 A Note

The user interface currently only runs properly on windows systems.