Building your own blockchain in R

Everybody is freaking out about the rise of the Bitcoin and the potential of the **Blockchain** technologies. The advent of cryptocurrencies, game changing use cases, disruption of established business models by disintermediation, etc..
By the time I'm writing this article, there are more than 1300 crypto-currencies listed in coinmarketcap.. And a lot more coming with the next ICOs (Internet Coin Offering).

Most certainly, the main enabler of Bitcoin and of many other currencies (although not all of them) is the Blockchain technology.

Although the original paper from Satoshi explains very well the ground idea behind this technology, nothing like **creating your own blockchain** to fully understand how it works, its limitations and potential improvements (aka ***"If you can code it, you certainly understand it"***).

In this post I'd like to share a gentle coding exercise in R (#rstats). Why R? Just because it's my favorite language… I wouldn't choose R for a productive, full-fledge block chain implementation, but again, this is not the purpose of this post. This is just a learning exercise hacked quite quickly without any aspiration of ever running this code in a productive environment, and should be understood as such.

For convenience, I stored the code in a git repository, for others to improve, re-use, try, etc.

First things first: what is what in a Blockchain

A **blockchain** is an immutable chain of sequential records or blocks that are "chained" together using hashes. **Blocks** can be understood as containers, typically of transactions, but it can be extended to documents, etc. We can think of a blockchain as a database where new data is stored in blocks, that are added to an *immutable* chain based on all other existing blocks.
Blockchain is often referred as a *digital ledger* of transactions performed in a cryptocurrency of choice (Bitcoin or whatever). A **transaction** requires a sender address, a recipient address and a given amount and needs to be assigned to a block.
The json below shows a typical block with a particular index, an integer representing the proof, a hashed representation of the previous block, which allows for consistency check (more about proof and hashing later) and a timestamp.

Once we have understood the concept behind the blockchain, let's build one and make it work :)
We are going to need three different files:

- The **class definition file**, where we create the Blockchain class with its components and methods.
- The **API definition file**, where we instantiate the class, register a node and expose the blockchain methods as GET and POST calls for the peers to interact with.
- The **plumber launch file**, to start the server and expose the API methods

Building a Blockchain

To represent the Blockchain, we need a list of blocks (the **chain**), a list of **currentTransactions** (waiting to be mined) and a list of mining **nodes**.

Our Blockchain class implements a method to register a new transaction **addTransaction**. Transactions are appended to the list of currentTransactions until a new block is created, which takes care of the newly added transactions that

have not been added to any block yet.
The creation of a new block takes place after calling **nextBlock**. To maintain the consistency, a new block can only be added knowing the hashed value of the previous one as part of a Proof of Work procedure (PoW). Basically, it is just finding a number that satisfies a condition. This number should be easy enough to be verified by anyone in the chain, but difficult enough so that a brute force attack to find it wouldn't be feasible (too long, too expensive).
In our example, **proofOfWork** relies on finding a number called *proof* such that if we append this proof to the previous proof and we hash it, the last 2 characters of the resulting string are exactly two zeroes. The way it works in the real Bitcoin chain is quite different, but based on the same concept.

In Bitcoin, the **Proof of Work** algorithm is called Hashcash. Solving this problem requires computational resources (the longer the strings and the more characters to be found within it, the higher the complexity), and miners are rewarded by receiving a coin—in a transaction.

The Blockchain class provides a method to check the consistency over all blocks **validChain**, which iterates over the chain checking if both each block is *properly linked* to the previous one and whether the PoW is preserved.
The method **registerNode** adds the URL of a mining node to the central nodes register.

Finally, as we need to think of a Blockchain as a distributed system, there is a chance that one node has a different chain to another node. The handleConflicts resolves this conflict by declaring the longest chain the proper one... the one that all nodes take.

Defining the API Methods

After defining the Blockchain class, we need to create an instance of it running on a node. First we generate a valid identifier for the node (using *Universally Unique IDentifier*). Then, we add the *genesis block* or first block in the chain, using some default parameters (previousHash=1 and proof=100).
Everything else takes place when users invoke the **"transaction/new"** method to create new transactions and miners call the **"/mine"** function to trigger the creation of new blocks according to the PoW schema and process the newly created transactions.

Apart from these core methods, we also enable the registration of new nodes **"/nodes/register"**, the consensus method to handle potential conflicts **"/nodes/resolve"**, the retrieval **"/chain"** and *html* display of the chain **"/chain/show"**.

To host these methods, we use rplumber, which is a wonderful package to expose R functions as *REST* and *RESTFULL* services. It's really versatile and easy to use (but single-threaded, which requires more advanced setups).
Apart from the aforementioned methods, we define a filter to log all requests with the server (*logger*).

We stored the code above in a file called "blockchain-node-server.R" to "plumb" it with the script below. First of all, we define a custom Serializer to handle a json boxing issue, as shown here.

Rplumber implements the swagger UI, which can be reached under *http://127.0.0.1:8000/__swagger__/*. The picture below shows our methods as we declared them in the blockchain-node-server.R script:

Interacting with our Blockchain

Once we have our Blockchain API running in the url we specified in the plumb command, we can try it with a simple client:

To try it locally, I opened 2 instances of RStudio: the first one for the server, where rplumber executes the server part, and the second one for the client, from where I fired all the sample requests from the previous script.
You can check the status of the chain in the browser, as shown in the picture below

But you can also interact with the chain programmatically from the client:

Credits

This coding has been inspired by different articles and tutorials (mostly in Python) showing how to build a blockchain from scratch. For example, the ones from Gerald Nash, Eric Alcaide and
Lauri Hartikka (this one in JS).
Special mention to the one created by Daniel van Flymen, very inspiring and easy to follow.