Report of the *Software Performance and Scalability* project

Sandro Baccega, Diletta Olliaro, Giacomo Zanatta, Giulio Zausa

Dipartimento di Scienze Ambientali, Informatica e Statistica
Università Ca'Foscari Venezia

A.A. 2019/2020

Contents

1 Introduction

This project needs to be contextualized in a performance evaluation setting, the problem addressed is to create a blockchain application that stores data on some US flights.
The main activity of the first part of this project is the implementation of the application itself, we can divide this task in a series of sub-tasks:

- defining transactions,
- definition of some operations such as:
  - adding a new transaction
  - retrieving a transaction given its ID
  - retrieving all the transactions in a specified block
- mining
- implementing the web application that must be able to:
  - adding a new record to the chain
  - query the status of a flight given its ID and the date
  - query the average delay of a flight carrier in a certain interval of time
  - given a pair of cities A and B, and a time interval, count the number of flights connecting city A to city B

This report will start with a first section explaining in detail the logic behind blockchains and then we will move on exploring all the sub-tasks presented above from a logical and implementation point of view.

2 Blockchains

A blockchain is a data structure that can store literally any kind of data, these data are stored in "blocks" that are linked together using cryptographic hashes.
The appealing properties of this data structure resides in the way this data is stored and added to the blockchain. In practice blockchain are nothing more than a linked list but with some constrains:firstly, once a block is added it is no longer modifiable, meaning that the only operation allowed on this "list" are appending and searching; secondly, there are specific rules for appending a new block that we will analyse in detail below; finally the architecture defined is distributed.

Each line of a dataset represents a so called *transaction*, a block contains one or more transactions. Each block, in order to be searchable and to guarantee some other nice properties, should have a unique ID. Since we also desire to avoid any kind of alteration in the data stored inside the block,we use cryptographic hash functions in order to "protect" the block.

At this point we need a way to be sure that our blocks are unmodifiable, this means we need a method such that a change in any block will invalidate the entire chain. In order to do this we create dependency among consecutive blocks by chaining them with the hash of the block immediately previous to them. Using this logic if we change the content of a block we also need to change its hash and consequently the previous hash field of the previous block, meaning we also need to change the hash of that previous block and so on for all the blocks before the one we wanted to modify.

This would not require an enormous amount of computational power if it was not the case that we also require a *proof of work* for any new block. What does it mean that we want a proof of work? This means that instead of accepting any hash for the block we add some constraint to it, in our case this constraint consists in the fact that we want a certain number of zero at the beginning of every hashes.
Main problem with this strategy is that unless we change the transactions in a blocks its hash will not change and of course we do not want to change transactions, so what we do is to add a new field to each block called *nonce*, this nonce is just a random number that will be used to compute the hash of the block together with the block itself. If the computed hash does not meet the imposed constraint we just need to increment it and so on until the produced hash is not as we want it. The nonce satisfying the constraint serves as proof that some computation has been performed.

The number of zeroes specified in the constraint determines the difficulty of our proof of work algorithm as a matter of fact the greater the number of zeroes, the harder it is to figure out the nonce. Consequently the proof of work is very difficult to compute but very easy to verify: once the nonce is figured out the only thing that needs to be done is to run again the hash function.