ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Systems@ETH Zürich

Autumn Term 2019 Prof. T. Roscoe, Prof. R. Wattenhofer

Computer Systems

Assignment 12

Assigned on: **December 9, 2019**

1 Authenticated Agreement

Quiz

1.1 PBFT: Basics

a) At which point of the agreement protocol can a node be sure that all correct nodes can only agree on the same request for a given sequence number within the current view?

b) During a view change, how can the backups be sure the new primary did not just make up requests that he wants them to execute?

c) During a view change, will only requests that were already executed by some correct node be included in the set $\mathcal{O}$?

d) It is possible that a node collected a prepared-certificate that will not be included in a new-view-certificate. Why is this not a problem?

Basic

1.2 PBFT: Utility of the Phases of the Agreement Protocol

In the PBFT agreement protocol, some phases seem superfluous. The purpose of this exercise is to get an insight to why those phases are necessary.

a) If correct nodes would not forward requests to the primary, how could a byzantine client slow down the system?

b) Assume that correct nodes do not wait for $2f+1$ `commit`-messages in the execute phase of the agreement protocol (Algorithm 25.17). Instead, they execute a request as soon as they have a prepared-certificate for it. How could it happen that two different correct nodes execute two different requests with the same sequence number?

Advanced

1.3 Authenticated Agreement

Algorithm 25.2 in the lecture uses authentication to reach agreement in an environment with byzantine processes.

a) Modify Algorithm 25.2 in such a way that it handles arbitrary input. Write your algorithm as pseudo-code. The processes may also agree on a special "sender faulty"-value. Hint: consider a set of values that correct nodes reached agreement for, then work with the size of the set.

b) Prove the correctness of your algorithm.

1.4 Multiple Prepared-Certificates for the Same Sequence Number!? — How can this happen?

One final corner case we did not consider in the script is implied by the answers to the quiz questions c) and d). In this exercise, we consider the possibility of multiple prepared-certificates for the same sequence number but different requests ending up in the $\mathcal{V}$-component of a `new-view`-message. How can it happen that the $\mathcal{V}$-component of a `new-view`-message contains two prepared-certificates, one for (v, s, r) and one for (v', s, r') with $r \neq r'$?

Remark: Exercise 1.5 asks you to show how this can be dealt with.

1.5 Multiple Prepared-Certificates for the Same Sequence Number!? — How can we fix it?

As we saw in Exercise 1.4, it is possible that multiple contradictory prepared-certificates for the same sequence number end up in the $\mathcal{V}$-component of a `new-view`-message. How does the primary have to choose between those prepared-certificates when generating $\mathcal{O}$ such that the system remains correct? Prove that your proposal guarantees correctness!

2 Advanced Blockchain

Basic

2.1 Randomness from previous Block

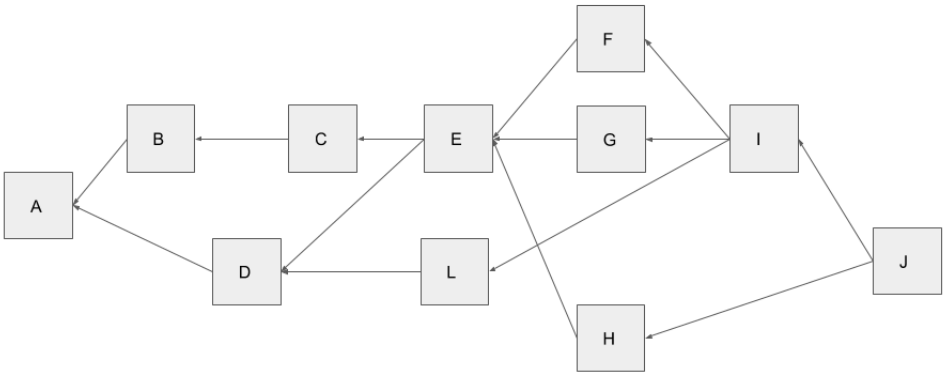
Consider the following (simple) Proof of Stake model: Assume there is a known set of validators who have each put up some currency as stake. For every block, the protocol needs to select one of these validators randomly. The probability of a particular validator being selected should be proportional to the amount of stake it has committed to the protocol.

Let's say the protocol selects the next validator as follows: We hash the previous block B and all its contents (transactions, metadata, timestamp, signatures, etc.) to get a hash value $h := h(B)$. Let's say the image of the hash function is $[0, H]$. We can partition this range into proportional sub-ranges based on the stake of each validator. Based on where h lands in this range, pick the validator whose sub-range overlaps with h to obtain the next validator.

How does this scheme devolve into proof-of-work?

2.2 DAG-Blockchain

Consider the following DAG-blockchain:



Arrows point from a child block to its parent blocks. Block A is the genesis block. Assume that the letters show the total order with respect to the blocks' hash values (this might be different from the DAG order).

a) What is the total order of block J's DAG-ancestors (the output of Algorithm 26.12 on block J)?

b) When creating the next block K, is there a way to assign its parents such that J's total order changes in the previous question?

Advanced

2.3 Selfish mining

The analysis of selfish mining shows that the selfish miner receives block rewards disproportionate to his hashing power share. Argue why this could be profitable in practice.

Remark: This is intended as an open question that might not have a definite answer. It is intended to spark discussion and encourage deeper understanding of the selfish mining problem.

2.4 Smart Contracts

Solidity is a high level programming language that compiles down to the EVM (Ethereum Virtual Machine). Solidity's documentation[1] has an example cryptocurrency smart contract. Modify this smart contract so that the creator can add more minters and any of them can mint more coins. Deploy this modified smart contract on the Ropsten Ethereum test network. A test network for Ethereum resembles the main network, but the Ether that is used in this network has no value. The test network is used for development/testing purposes. Through the smart contract ABI (Application Binary Interface), invoke the call to add another minter, and a follow up call to let this new minter mint new coins for another unrelated address.

A few points:

- You can either mine "fake" Ether on the Ropsten network, or request for Ether from a test faucet. Or request from some special websites.
- You can either run an Ethereum full node connected to the Ropsten network, or you can connect to a third party node that is run by others, like Infura or QuikNode.
- You can either compile/deploy the smart contract from scratch, or you can use a framework like Truffle.

[1] `https://solidity.readthedocs.io/en/latest/introduction-to-smart-contracts.html#subcurrency-example`