Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Systems@ETH Zürich

Autumn Term 2020 Prof. T. Roscoe, Prof. R. Wattenhofer

Computer Systems

— Solution to Assignment 12 —

1 Authenticated Agreement

Quiz

1.1 PBFT: Basics

a) According to Lemma 25.18, it is impossible that two prepared-certificates for the same sequence number are gathered within the same view (not even at different nodes). Therefore, once a node has obtained a prepared-certificate for a request r in view v, it can be sure that no correct node will execute a different request $r' \neq r$ with the same sequence number in view v, as this would also require a prepared-certificate for the other request within the same view.

b) The new primary has to send around the new-view-certificate $\mathcal{V}$; that certificate has to be valid and the set of `pre-prepared`-messages $\mathcal{O}$ has to be constructed validly from $\mathcal{V}$ in the way specified by the protocol. Since $\mathcal{V}$ already determines the content of $\mathcal{O}$ and the `view-change`-messages in $\mathcal{V}$ are signed, correct replicas can rely on $\mathcal{O}$ if the above conditions hold.

c) Not necessarily. It is possible that some node u collected a prepared-certificate for a triple (v, s, r), but as soon as u collected the prepared-certificate, a view change happened. In that case, no correct node can have executed that request yet, but u's `view-change`-message could still end up in the set $\mathcal{V}$ of the `new-view`-message for the next view.

d) The proof of Theorem 25.26 shows that if a request was executed by a correct node, then a prepared-certificate will end up in $\mathcal{V}$. If we take the contrapositive of that statement, we find that if there is no prepared-certificate for a request in $\mathcal{V}$, then no correct node has executed that request yet. Omitting prepared-certificates for requests that no correct node executed cannot harm correctness of the system.

Basic

1.2 PBFT: Utility of the Phases of the Agreement Protocol

a) Backups start their faulty-timer after they receive a request. If backups do not forward requests to the primary, then a faulty client could just send requests to the backups, and the backups' faulty timers would permanently keep expiring, inducing view change after view change.

A byzantine client could make sure to send a request to a backup even without knowing which node is the primary by simply sending distinct requests to all nodes; all but one node will be backups, and all of their faulty-timers would start running for requests that the primary has never seen and for which the primary can therefore not start the agreement protocol.

b) Lemma 25.18 implies that two correct nodes cannot agree to execute different requests within a single view, and the proof does not rely on nodes waiting for `commit`-messages, so this Lemma remains intact even with the alteration made in this exercise.

However, the `commit`-messages are important for the view change protocol to maintain safety across views, which we can see in the proof of Theorem 25.26. Consider the following sequence of events:

1. Node u collects a prepared-certificate matching (v, s, r), and directly executes r. No other node has seen a prepared-certificate yet, and a view change occurs at this moment.
2. The new primary p' of view $v' > v$ collects $2f + 1$ `view-change`-messages, and u's message is too slow to be included. p' thus does not add a $\texttt{pre-prepared}(v', s, r, p')_{p'}$-message to $\mathcal{O}$.
3. In the new view v', correct nodes (with the "help" of byzantine nodes) run the agreement protocol for (v', s, r') for some $r' \neq r$. As soon as correct node $w \neq u$ collects a prepared-certificate matching (v', s, r'), node w will execute r' with sequence number s.

With this, u will execute r with sequence number s, and w will execute $r' \neq r$ with sequence number s.

If $s < s^{\mathcal{V}}_{max}$ (cf. Algorithm 25.24), then r' will be `null`. However, if $s > s^{\mathcal{V}}_{max}$, then r' can be a distinct non-`null` request.

Advanced

1.3 Authenticated Agreement

a) We can do roughly the same as we did in Algorithm 25.2, but for multiple values in parallel. Every backup will be collecting messages for every value they hear about. If a correct node gathered agreement for multiple values (or for no values) after $f + 1$ rounds, then it knows that the primary must be faulty. The new algorithm can be seen in Algorithm 1.

b) The proof is very similar to the one in the script, so we will only give a rough sketch of how to adapt it here:

- If the primary is correct, then he only sends one message $\texttt{value}(x)_p$ in the first round, and all correct backups decide on x after round $f + 1$.
- If the primary is byzantine, then there are these cases:
  1. No correct node ever adds a value to A, then all correct nodes output "sender faulty".
  2. (The proof of this case is analogous to correct nodes deciding on 1 in the proof in the script. Check the proof in the script if some detail here is unclear.)
     At least one correct node adds at least one value x to A. For any value x that gets added to A by some correct node, the first time a correct node adds x to A necessarily happens in a round $i < f + 1$, and all correct nodes will have $x \in A$ in round $i + 1 \leq f + 1$. Since this holds for all x, all correct nodes have the same A after round $f + 1$.

Algorithm 1 Byzantine Agreement with Authentication

Code for primary p:

1: $x \leftarrow$ input value of p
2: broadcast $\texttt{value}(x)_p$
3: decide x and terminate

Code for backup b:

4: $A \leftarrow \emptyset$
5: **for all** rounds $i \in \{1, \ldots, f+1\}$ **do**
6: **for all** messages $\texttt{value}(x)_u$ that b received this round **do**
7: $V_x \leftarrow \{$all messages $\texttt{value}(x)_v$ that b received since round $1\}$
8: **if** $|V_x| \geq i$ and $\texttt{value}(x)_p \in V_x$ **then**
9: $A \leftarrow A \cup \{x\}$
10: broadcast $V_x \cup \texttt{value}(x)_b$
11: **end if**
12: **end for**
13: **end for**
14: **if** $|A| = 1$ **then**
15: decide on the single element in A and terminate
16: **else**
17: decide "sender faulty" and terminate
18: **end if**

If A contains exactly one value after round $f+1$, then all correct nodes decide on that value, otherwise all of them decide on "sender faulty".

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

In Quiz question c), we saw that a correct node can collect a prepared-certificate that will not be included in a `new-view`-message. Working from this insight, we can imagine the following sequence of events:

1. Only correct node u collects a prepared-certificate for (v, s, r).
2. A view change to view $v+1$ happens, and the prepared-certificate that u collected for (v, s, r) is not included in the view change.
3. Correct node w collects a prepared-certificate for $(v+1, s, r')$ for some $r' \neq r$.
4. A view change to view $v+2$ happens. Both prepared-certificates are included in the `new-view`-message.

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

Notice that step 2. of the solution to Exercise 1.4 can only occur if the prepared-certificate that u collected was for a request that no correct node executed, see the solution to Quiz question d). This means that we can ignore that prepared-certificate without worrying about it.

On the other hand, for every request that was executed by some correct node, a prepared-certificate for it will end up in every subsequent `new-view`-message, see the proof of Theorem 25.26.

From these considerations, we define the protocol to do the following in Algorithm 25.24: if multiple prepared-certificates for sequence number s end up in $\mathcal{V}$, then let v^* be the highest view number for which a prepared-certificate (v^*, s, r) exists in $\mathcal{V}$. The primary p of new view v will include a $\texttt{pre-prepare}(v, s, r, p)_p$-message in $\mathcal{O}$ and ignore all other prepared-certificates for s. The claim we now make is this:

Theorem 1. *Let v^* be the maximum view number of any prepared-certificate for s in $\mathcal{V}$. If the $\mathcal{V}$-component of a* `new-view`*-message contains multiple prepared-certificates for the same sequence number s, then due to Lemma 25.18, there is at most one such prepared-certificate per view v^*. While constructing $\mathcal{O}$ in Algorithm 25.24 during a view change, let the primary of the new view only include a* `pre-prepare`*-message for the prepared-certificate matching s with the highest view number among the prepared-certificates for s in $\mathcal{V}$ (this is the* latest *prepared-certificate for s). Then, if some correct node executes a request r with sequence number s in view v, then the latest prepared-certificate for s in $\mathcal{V}$ during every view change after v will match (s, r).*

Proof. If no `new-view`-message ever contains two prepared-certificates for the same sequence number s but different requests $r' \neq r$, then the Theorem is already proved in the proof of Theorem 25.26 in the script. If no correct node ever executed a request at some sequence number s, then there cannot be a problem with correctness at s either. Thus, assume that a correct node executed request r with sequence number s in view v, and some subsequent `new-view`-message contains multiple prepared-certificates for s. We prove the theorem by induction, showing that once a correct node executed r with sequence number s in view v, then all prepared-certificates for sequence number s that nodes collect in later views will match (s, r).

Base case: Consider the first view $v' > v$ in which the new primary p' sends a valid `new-view`-message. Since correct backups reject invalid messages, no correct node entered any view $v^\dagger$ with $v < v^\dagger < v'$, so no prepared-certificates were collected in any such view $v^\dagger$. Thus, v is the highest view number for which any node collected a prepared-certificate for s. As shown in the proof of Theorem 25.26, a prepared-certificate matching (v, s, r) will be in $\mathcal{V}$ in the `new-view`-message for v'.

Induction step: Consider a view change from v' to v'' with $v < v' < v''$ and assume that up to v', the latest prepared-certificate in the $\mathcal{V}$-component of `new-view`-messages for s has matched (s, r). Because backups respond to $\mathcal{O}$ in Algorithm 25.25 before responding to any other `pre-prepare`-messages, nodes can only have collected a prepared-certificate for s with the same r during view v'. Thus, during the view change from v' to v'', the latest prepared-certificate for s that is in $\mathcal{V}$ will match (s, r) as well. □

Notice how this clarifies the answer to Exercise 1.4: it can happen that multiple prepared-certificates for the same sequence number s exist in a `new-view`-message. However, if the new primary always picks the latest such prepared-certificate to react to when constructing $\mathcal{O}$, then that guarantees that once a request r was executed at s by any correct node, then no node will ever be able to gather a prepared-certificate for (s, r') with $r' \neq r$. Thus no correct node will ever execute anything but r at sequence number s.

2 Advanced Blockchain

2.1 Randomness from Previous Block

The validator who has been assigned the hash range starting from `000000...` to, say, `000000FFFFFF...` has an incentive to construct a block such that the hash of that block will land in the their range of hash. In other words, they control the randomness of the process. In this way, they will iterate over many combinations of block content until they find the right block that will result in a specific hash that is favorable to them in the next round. QED.

2.2 DAG-Blockchain

a) *A B C D E F G L I H J*

b) No. Since none of the nodes *A*–*J* have any reference to K (otherwise it would not be a DAG), the total order for all of *K*'s parents remains unchanged by the creation of *K*.

2.3 Selfish mining

Here's our (somewhat subjective and non-conclusive) answer to this question:
Until the next difficulty adjustment, the block generation rate with respect to the longest chain is reduced in the presence of a selfish miner as the total hashing power would be distributed on two forks of the chain. Here, the selfish miner might even get unlucky and make less profit than by following the protocol.

However, due to the difficulty adjustment, the rewards distributed among the miners finding blocks will be reestablished to match a constant rate. From Theorem 26.3 we know that it is rational (profitable) for a miner to mine selfishly if it has enough hashing power (e.g. $\alpha \geq \frac{1}{3}$ or even some smaller α with $\gamma > 0$).

Further thoughts:

- Assume that miners reinvest some of their income in buying new mining equipment. As the global rewards per hour are kept at a constant level due to the difficulty adjustment and a selfish miner would get disproportionally high rewards, the honest miners must be awarded disproportionally small rewards. This would mean that the selfish miner will have a greater increase in his hashing power in comparison to the honest miners. Consequently, his share of block rewards will increase even more.
- Assume that other miners are rational (miners trying to maximize their own profit). They would be better off joining the selfish miner. This poses the potential for a majority attack.

Remark: There are papers that compare selfish mining rewards vs. honest mining rewards over a longer term taking Bitcoin's difficulty adjustment into account. For example: `https://arxiv.org/abs/1912.01798` *and* `https://eprint.iacr.org/2018/1084.pdf`.

2.4 Smart Contracts

In our opinion, it's easiest to use the following tools for this example smart contract. YMMV.

- Use Metamask to get a Ropsten test network address.
- Use `https://faucet.ropsten.be` to get Ether into your Ropsten address.
- Use Ganache to instantiate a local blockchain.
- Use Truffle to build/test the smart contract on the local blockchain.
- Use Truffle/Infura to deploy the smart contract on the Ropsten network.

The following is the diff between the example smart contract and the one with additional minters:

```
7a8
>     mapping (address => bool) public additional_minters;
22c23
<         require(msg.sender == minter);
---
>         require(msg.sender == minter || additional_minters[msg.sender] == true);
24a26,30
>     function add_additional_minter(address additional_minter) public {
>         require(msg.sender == minter);
>         additional_minters[additional_minter] = true;
>     }
```

The above tools show that the actual building/deploying of smart contracts on Ethereum is quite straightforward. Smart contracts themselves can be complicated and need to be extensively reviewed/tested/formally verified. Many third party libraries provide standard functionality that is tested extensively beforehand. Most of the times, it's recommended to use these libraries. But sometimes, it can be quite catastrophic - see `https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/`.