Homework 3
Naming with a Blockchain

CS-438 - Decentralized Systems Engineering, Fall 2020

Publish date: Tuesday, November 17, 2020
Due date: Tuesday, December 8, 2020 @ 23:55

General Guidelines

In this homework, you will continue to build more functionalities to *Peerster* **on top of your Homework 2 implementation**. In this homework, you will add functionalities to achieve globally consistent file naming through a blockchain.

We will provide you with the skeleton files that contain the main interfaces to be implemented, test framework, and some useful functionality and implementation hints. The skeleton files are implemented on top of the skeleton for Homework 2, which means that you should be able to merge it with your existing implementation. Specifically, you will receive access to a new repository cs438-hw3-student-*name* which you can add as a new git remote in the folder with your existing implementation and pull the updates into your master branch, e.g., by running

```
git remote add hw3
https://gitlab.epfl.ch/cs438/students/cs438-hw3-student-XYZ.git

git pull --rebase hw3 master
```

Then you should be able to merge the master branch into the branch with your implementation. If you face merge conflicts, you can resolve them either manually, by opening the files with the conflicts and removing the unnecessary code pieces, or using an automatic tool, such as KDiff3, (it can automatically resolve the conflicts of modifying the same file as long as the changes do not touch the same source lines), although you will still likely have to manually resolve some of the conflicts, for which you need to use your own reasoning.

Note that you *must* use the new hw3 repository (`cs438-hw3-student-`*`name`*) to work on this homework, i.e. to make new commits and run tests.

Throughout this development and debugging process, you are welcome to discuss challenges and techniques with your fellow students, exchange pointers to relevant information or algorithms, debugging tips, etc., ***provided you each write your own code independently*. Homeworks are individual per student.**

Teaching assistants will be available on Zoom every Friday 15:15-17:00, to discuss with you how to architect your implementation. ***TAs are not going to debug your code***, but they can

help you ask the right questions just like your software engineer colleagues will do in the future. The room INF 1 / the timeslot 13:15 - 15:00 (depending on whether access to the campus is allowed) is available every Monday from 13:15 to 15:00 for you to hack together and test your implementations, without TA supervision.

Fourth Life, a.k.a. Code Review

Every homework builds on the previous one and requires a full and working implementation of the previous homework. Thus, we strongly encourage you to fully implement every homework. However, if it so happens that: (1) You were unable to complete a homework assignment, or (2) You fully implemented your homework but the poor code design makes it hard for you to build on top, or (3) You are unsure why you don't pass all tests, we offer you an alternative.

After the deadline of Homework 2, you will receive 3 random submissions of your colleagues (by getting read-only access to their HW2 repositories on Gitlab). You can choose to build on top of any of these 3 assignments for your next homework. **If you decide to do so, you need to specify that homework's identifier, i.e., the Gitlab repo link, in your future homework submission on Moodle.** Also, if you build on someone else's code, we *strongly suggest* that you review that colleague's code on Gitlab as a token of appreciation -- the review is, however, not graded.

However, be warned that there is no guarantee regarding the quality of these 3 randomly assigned submissions - they might be less good than your own implementation. Your best strategy, thus, is still to complete the homework yourself.

Introduction

Now that we can search for files, we need a human-readable identifier for each file instead of using the metafile hash. We also need this *name-to-metahash mapping to be global and agreed upon* to protect from adversarial peers who serve malformed/malicious files instead of the correct ones. A blockchain enables us to reach our goal.

You will enhance Peerster peers with the following:

1. Claiming the mapping of file names to their metafile hashes
2. Processing the claimed mappings of other peers
3. Publishing the accepted mappings to a blockchain

A claim for mapping a file name to a metahash is nothing else than a transaction. So now, whenever you add a new file to your Peerster to be indexed and potentially shared, you need to notify other peers in the network about this exciting event by sending out a corresponding transaction. The nodes add this transaction to the blockchain if they reach consensus on the file name to metahash mapping.

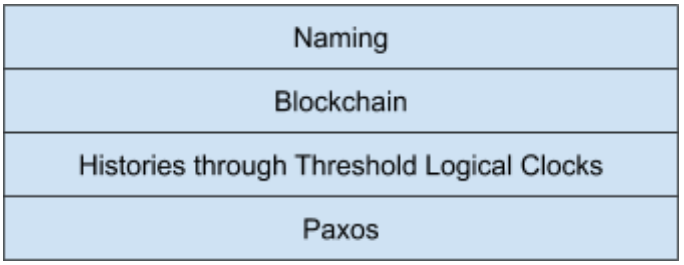

Figure 1: The stack layer

Architecturally, the assignment has four layers:

1. Naming: its responsibility is to expose the mapping between filenames and file metahashes to the application. To do this, it relies on the Blockchain.
2. Blockchain : the blockchain represents an ordered succession of blocks, where each block contains its number, a reference to the previous block, the file name and metahash that the majority agreed upon. This is essentially a distributed ledger, built incrementally and sequentially by consensus. The sequencing is the responsibility of the next layer.
3. Histories through Threshold Logical Clocks : the algorithm we use to advance sequentially from one block to another in the blockchain is a variant of Threshold Logical Clocks (TLC). TLC is a primitive where the logical clock of a node - here representing the block number - advances when the logical clocks of a majority of other nodes also advance. When do we advance? When a consensus has been reached, which derives from the last layer, Paxos.
4. Paxos: each block in the blockchain represents the decision of a consensus execution, which we implement using a Paxos consensus box (i.e. an instance of Paxos, where nodes want to agree on a single value) for achieving consensus on a single file name.
   When multiple nodes make simultaneous claims, i.e., several filename-metahash associations are competing for the same block number, all these claims are inputs for the same consensus box. The output is the claim that the network reached consensus on for that particular Paxos consensus box.

We make the simplifying assumption that, throughout the execution, the set of participating nodes does not change, except for a minority **f** of nodes that may fail in a crash-stop manner. In particular, every node knows from the beginning the number of participating nodes **N** (though it may not directly know all participating nodes). We assume that f is strictly smaller than a majority, otherwise consensus is impossible, but we do not give a particular value for f beyond this assumption.

Part 1: Paxos (consensus on a single filename)

We'll use the Paxos consensus algorithm, which should be familiar to you from the lecture. For consensus, every node in Peerster plays three roles: (i) *Proposer* as a client that submits transactions; (ii) *Acceptor* as a node that accepts or rejects transactions, and (iii) *Learner* as a node that learns the outcome of the consensus. As you may know, Paxos might require

several rounds to complete, but all these rounds are part of the same consensus box and all have the purpose of achieving consensus on a single filename-metahash mapping.

The simple case

This section describes the simple case where there is no contention among proposers, i.e., a single proposer proposes a value, and there are no failures. The other cases are described in the next section.

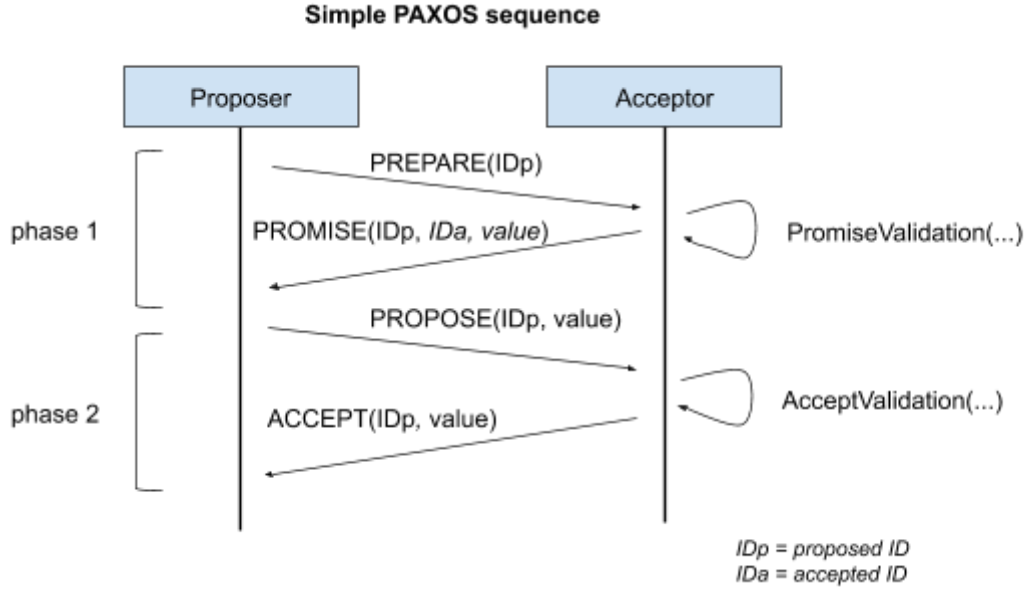

Figure 2: Sequence of Paxos messages on a simple case between a *Proposer* and an *Acceptor. IDs explained in the next section.*

When a node shares a file (as per Homework 2), it also sends out a claim for a file name. There are two phases in the Paxos protocol: **prepare/promise** and **propose/accept**.

- First, the proposer node creates a name claim transaction and gossips it with the entire network by sending a PREPARE message. Every node receiving the claim runs a `promiseValidation` function to either consent with or reject the claim and, **if it consents to it**, correspondingly sends a PROMISE back to the proposer. The proposer waits for a majority ⌊N/2+1⌋ of PROMISE and, if all are promises **for its own value**, starts the second phase (more on other cases below).
- In the second phase, the proposer node gossips the same claim as a PROPOSE message. Every node receiving the PROPOSE runs an `acceptValidation` function and, **only if accepted**, it sends an ACCEPT to all nodes (because all the nodes are learners).

All nodes wait for a majority ⌊N/2+1⌋ of ACCEPT to accept messages for the same value. When that happens, the nodes know that the algorithm reached consensus on that value and complete the current Paxos execution.

Contention and failure cases

To guarantee safety in case of node failures or contention between proposers, Paxos introduces rounds, which help nodes order the proposals and reason on whether to promise (phase 1) or accept proposals (phase 2). For safety, rounds need to be unique across nodes. Every node maintains a local variable `ID`, unique across the system. We implement it as follows: the `ID` is initialized with `nodeIndex` (passed to the gossiper via a CLI flag). Throughout the document, incrementing the `ID` means obtaining the **next unique ID value**, which is simply the old `ID` plus N, the number of participating nodes.
In the PREPARE and PROPOSE messages, the proposer adds its `ID` to each of those two messages. Acceptors remember the highest `ID` they have seen.

In the `promiseValidation` function, a node acting as an acceptor responds with a PROMISE if both of the following hold:

- the acceptor has not seen any other PREPARE with a higher or equal ID **and**
- the acceptor has not yet accepted any proposal (in the current Paxos run), otherwise, it replies with a PROMISE containing **the ID and value of the previously accepted proposal**

In the `acceptValidation` function, a node acting as an acceptor responds with an accept if:

- the proposal's ID equals or is greater than[1] the highest ID the acceptor has promised to any proposer (in the current Paxos run)

The proposer waits for a majority ⌊N/2+1⌋ of PROMISE replies. If the proposer does not receive a majority ⌊N/2+1⌋ of positive promises within a timeout of `paxosRetry` seconds (passed through a CLI flag), and its node (in its quality of learner) has not reached consensus in the current Paxos run, then the proposer increments its `ID` and tries again. Otherwise, the node starts phase 2, either with its own value or, if any of the replies contain an accept for another value V, the proposer starts phase 2 with its own `ID` but with the claim V.

Similarly, the proposer node waits for a majority ⌊N/2+1⌋ of replies to the second phase within a timeout of `paxosRetry` seconds. If it observes a majority ⌊N/2+1⌋ of ACCEPT messages, it stops. Otherwise, if the node, in its quality of learner, has not reached consensus in the current Paxos run, then the proposer increments its `ID` and tries again the whole Paxos procedure starting from phase I with a PREPARE message.

[1] In some pseudocodes for Paxos (including the one in Reference 4 at the end of the handouts), the condition is strictly "equals". Those pseudocodes might assume reliable in-order delivery of messages, which means that a node wouldn't observe a PROPOSE before its corresponding PREPARE. In Peerster we have reliable delivery but for simplicity we don't enforce ordering, thus the condition is "greater or equal".

Implementation

The skeleton contains the basic structures for the implementation. The structure `Block` represents a file name claim, which contains the name, the metahash, the block number and hash of the previous blockchain block. For Part 1, the block number should be 0 and the previous hash should be an empty array `[]byte`. When sending a message in any of the protocol phases, you add a `Block` to your message.

The skeleton also defines the message types for all phases of the protocol: `PaxosPrepare`, `PaxosPromise`, `PaxosPropose`, `PaxosAccept`. All these messages are sent as rumors and they're embedded in the rumor using a new optional field, called `ExtraMessage`.

You'll notice that acceptors send back PROMISE messages embedded in Rumor messages. Normally, the PROMISE would be implemented using unicast messages, i.e., private messages in Peerster; but, because private messages are unreliable, we chose for simplicity to use rumors that have reliability in place thanks to status messages. Essentially, we trade off bandwidth (broadcast instead of unicast) for simplicity.

To increment the `ID`, you can use the `UniqueIDGenerator` interface, provided in the skeleton.

Gossiper flags

The gossiper receives the following flags additionally to the ones from previous homeworks:

- `-N` flag for the number of participating nodes
- `-nodeIndex` flag for initializing the node's ID
- `-paxosRetry` the timeout in seconds to wait for a reply before retrying Paxos

Client

We do not provide a GUI for this homework. For the CLI client, please use the homework 3 client that adds a file for sharing. This should trigger consensus on file naming.

Watchers

The watcher should still work as before: every message sent/received from/to the gossiper should be notified to the watchers. We are expecting to see, among others, the Rumor messages that contain, in their Extra fields, the Paxos messages.

Part 2: Blockchain

All nodes, as learners, maintain a blockchain locally and add the consensus value to their blockchain. Eventually, all live nodes learn about the consensus, as there is a guarantee that rumors are eventually delivered[2].

[2] Remember? You implemented this with Status packets and their associated status-sharing logic!

However, it would be futile to start executing a new Paxos box until a majority ⌊N/2+1⌋ of nodes have learned about consensus in the current Paxos box. Thus, nodes should start executing a new Paxos box, and implicitly be ready for the next block, when a majority ⌊N/2+1⌋of other nodes also **know** that consensus has been reached in the current Paxos box. To implement this, we'll use a primitive akin to Threshold Logical Clocks (TLC).

The main idea behind TLC is to progress at the speed of the majority ⌊N/2+1⌋, and in our case, at the speed of a majority ⌊N/2+1⌋of nodes observing the consensus. When a node observes consensus, it needs to notify all the other nodes of this event.
A node observes a consensus when it receives a majority ⌊N/2+1⌋of ACCEPT messages (Paxos's safety ensures that all ACCEPTs are for the same value) in the current Paxos box. To notify others, the node broadcasts a TLC message that contains the `Block`.
When a node collects a majority ⌊N/2+1⌋ of TLC messages from the current Paxos box, indicating that a majority ⌊N/2+1⌋ of nodes observed consensus on a certain value, the node knows that it can fast forward the decision of its current Paxos box and proceed to the next block by starting to execute a new Paxos box, even if it has not observed consensus on its own.

If the consensus for a certain block number was reached on a different value than the mapping proposed by a node, the node should immediately retry unless its proposed mapping clashes with the already accepted one (i.e.., the node wants to claim a filename that already exists in the blockchain).

Implementation

The implementation is rather straightforward. To keep track of the current block number in the blockchain, every node maintains a local variable `block_number`. When the node advances its logical clock and implicitly starts executing a new Paxos box, the node increments its `block_number` variable and resets the `ID` to nodeIndex. Pay attention that IDs and block numbers are different concepts: IDs are within the same Paxos box, whereas blocks correspond to different Paxos boxes.

For the first block, the previous hash should be a byte array filled with 32 zeros: `00...0`. We use this value to distinguish it from a hash referring to an empty block.

To broadcast TLC messages, a `TLCMessage` should be embedded in a Rumor. Specifically, it's embedded using the same new optional field, called `ExtraMessage`.

You will need to use the exact implementations of hashing provided in the skeleton in order to be compatible with other implementations of the blockchain.

Gossiper flags

The gossiper receives the same additional flags as Part 1:

- `-N` flag for the number of participating nodes
- `-nodeIndex` flag for initializing the node's ID
- `-paxosRetry` the timeout in seconds to wait for a reply before retrying Paxos

Client

We do not provide a GUI for this homework. For the CLI client, please use the homework 3 client that adds a file for sharing. This should trigger consensus on file naming.

Watchers

The watcher should still work as before: every message sent/received from/to the gossiper should be notified to the watchers. We are expecting to see, among others, the Rumor messages that contain, in their Extra fields, the Paxos **and TLC** messages.

References

1. The course's lectures relevant for consensus and blockchains
2. "Paxos Made Simple", Leslie Lamport.
3. "Paxos made live: an engineering perspective", Tushar D. Chandra, Robert Griesemer, Joshua Redstone.
4. Understanding Paxos, Paul Krzyzanowski. Blog post from Rutgers University.