CS-438 - Decentralized Systems Engineering, Fall 2023

Homework 3

Consensus

Consensus is tricky (...)

- B. Lampson

Author	DEDIS lab, EPFL
Revision	September 2023 - 1.1.0
Publish date	Monday, November 6, 2023
Due date	**Tuesday, December 5, 2023 @ 23:55**

Table of contents

Introduction

End result

The goal of this homework is to agree on filenames to metahash associations. To this extent, peers will run several instances of consensus, with each instance agreeing on an association.

You are going to revisit and improve the `Tag()` function from the data sharing module you implemented in the previous homework. You will ensure that once a file (i.e metahash) has been named in the system, all peers see it under the same name, effectively ensuring that all name stores are consistent and synchronized across all peers in the system. This functionality will be implemented by building a very simple blockchain.

The `Tag()` function will allow a peer to associate a name to a metahash across the whole Peerster system. Once a file is tagged by a peer, a Paxos consensus instance will be initiated to propagate the association to all peers and resolve any conflict if several peers concurrently propose other name/metahash associations. An instance of the Paxos consensus will associate a single metahash to a file name, so a multi-Paxos implementation will actually be needed to support a series of consensus instances covering multiple metahashes. As a performance optimization, Threshold Logical Clocks (TLC) will be used to speed up the propagation of consensus decisions. Finally, each agreed upon name/metahash association will be stored in a block, and the blocks will be chained as part of a blockchain.

In the rest of this document we refer to “value” as the name/metahash association. As you will discover later, this value is defined by a type.

A typical workflow will be:

1. Tag an uploaded file with a filename N that maps to a metahash M.
2. The Paxos consensus protocol runs to agree on (N,M) mapping and the peers agree on a single mapping.
3. A new block is created and the Threshold Logical Clock moves forward.
4. The block is added to the blockchain, and the name store is updated.

Objectives

Improve the `Tag()` function from the datasharing module of homework 2.

Paxos

As part of this homework, you will have to be familiar with the Paxos algorithm, whose description goes beyond what can be pragmatically described in a handout. Before starting this assignment, we encourage you to carefully read the following paper, which is a compulsory course reading, to have an idea about the Paxos algorithm:

Paxos Made Simple https://lamport.azurewebsites.net/pubs/paxos-simple.pdf

In order to ensure consistency, the implementation is going to use the same terms and protocol as described on the following webpage:

Understanding Paxos https://people.cs.rutgers.edu/~pxk/417/notes/paxos.html

Paxos in Practice

To integrate Paxos within the Peerster system, you need to adhere to the following specifications:

1. Paxos requires proposal numbers (IDs)[1] to be globally unique. For that purpose each Peer is provided with a unique ID, along with the total number of peers N in the system.
    a. The starting ID of each Paxos proposal will be the proposing peer's unique ID. This unique ID is distributed among peers from 1 to N (1 for the first peer, 2 for the second peer, etc…) using a configuration parameter.
    b. If a peer needs to make a new proposal with a higher ID, it increments its previously used ID by the total number of peers N. As such, the starting ID of the first peer will be 1, and its next IDs will be 1+N, 1+2*N, and so on.
    c. Note that IDs are scoped to a Paxos consensus instance.
2. The Paxos `PROMISE` message will be broadcasted in a private message. All the other messages (`PREPARE`, `PROPOSE`, `ACCEPT`) will be directly broadcasted.
3. When we mention the broadcast, we refer to the broadcast function you implemented in HW1.
4. Effectively all peers act as acceptors and learners, and can be proposers.

[1] Throughout this document we use the term ID to denote the concept of *Ballot* in the original Paxos paper. We follow the terms from the website *Understanding Paxos*.

Multi-Paxos

An important aspect of this homework is that you will deal with repeated uses of Paxos, as you will use Paxos to agree on multiple values. In the following explanation, we refer to each of the repeated executions of the Paxos protocol as an “instance” of Paxos. We also see multi-paxos as a sequence of steps, where each step is the execution of a paxos instance. Step 0 refers to the first Paxos instance. The usage of multiple consecutive Paxos instances is typically called multi-Paxos and carries its own set of challenges.

In particular, every peer in the system, upon receiving a Paxos message, needs to be able to understand if the message belongs to a past, present, or as-of-yet-unknown instance of Paxos. This allows the node to know when the message can be discarded and when it needs to be processed. It is specifically expected that proposer and acceptor peers would drop messages unrelated to the current multi-paxos step

Threshold Logical Clocks (TLC)

To advance sequentially from one Paxos instance to another, as fast as a quorum of peers allows it and robustly, we can make use of Threshold Logical Clocks (TLC). TLC is a design pattern for threshold communication, and the formulation in this homework is just one of many ways to run multi-paxos. TLC is a primitive which enables an asynchronous and distributed system to have a seemingly synchronous behavior as seen from the layers above it. It does so by making use of a logical clock derived from the underlying messaging among various peers.

The logical clock of a peer advances when the logical clock of a quorum of other peers is ready to advance. The TLC logic is thus pretty simple: the clock is either waiting to reach a threshold[2] of TLC messages to advance, or advancing to the next step[3]. It does so for all steps.

It is thus important to keep track of all the TLC messages received for both the current clock value and future clock values, so that the clock can move forward as soon as a threshold is reached. Conversely, TLC messages referencing past clock values can be dismissed, as they will not affect the logical clock.

In our case, a peer broadcasts a TLC message in two cases:

1. When a consensus is reached at the Paxos layer, or
2. When a peer receives a quorum of TLC messages for the current clock. Note that the peer must broadcast a TLC message only once per clock.

[2] *Quorum* and *Threshold* refers to the same concept in this document: a *quorum* of peers is the same as a *threshold* of peers.
[3] *Step* and *Clock* are the same concept in this document: when a peer *moves to the next step*, it indeed *moves its clock*.

The logical clock effectively represents the multi-paxos step number: when TLC moves to the next clock, a new paxos instance takes place which makes them both move at the same pace synchronously. TLC should be responsible for moving multi-paxos to the next instance, while the paxos instance is responsible for producing the accept messages that TLC uses to know when a consensus has been reached. Figure 1 illustrates how the logical clock goes to the next step and the relation it has with multi-paxos. In this figure, each paxos instance corresponds to a TLC step.

Since the communication is based on rumors, we can be sure that all peers, even those joining late, will eventually receive all messages. This makes it possible for “late joiners” to quickly catch up the state of the application by going through TLC messages, without having to process the Paxos messages themselves or be involved in the consensus.

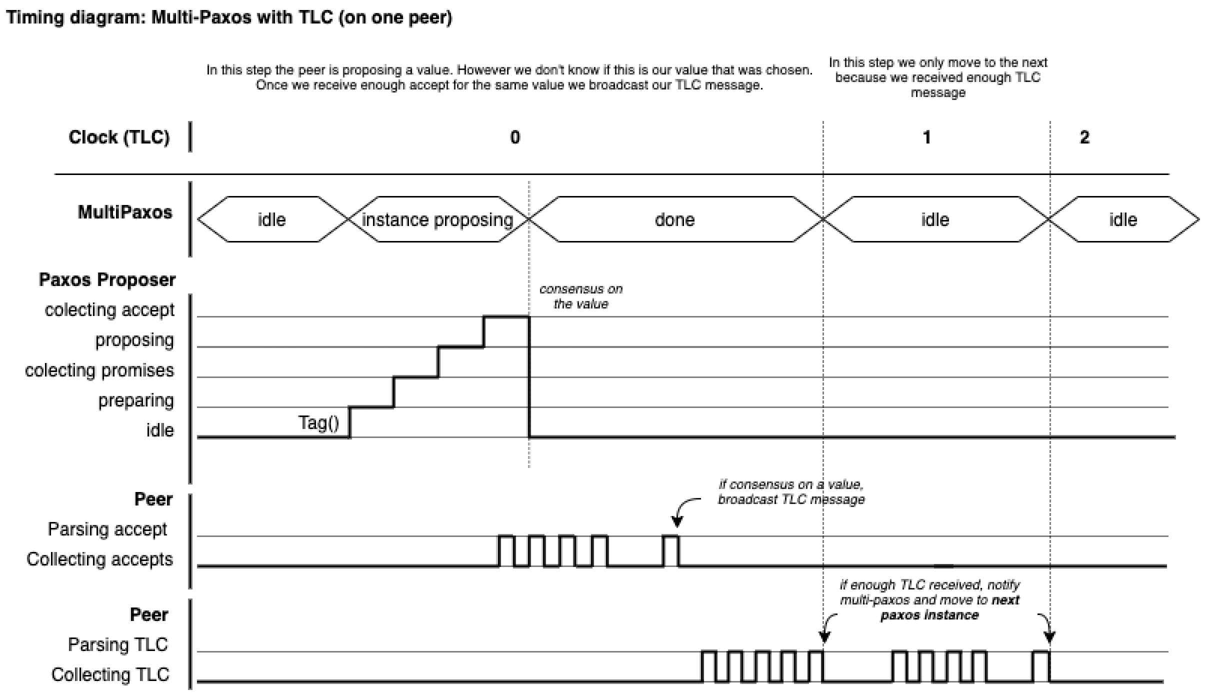

Timing diagram: Multi-Paxos with TLC (on one peer)
In this step the peer is proposing a value. However we don't know if this is our value that was chosen. Once we receive enough accept for the same value we broadcast our TLC message.
In this step we only move to the next because we received enough TLC message
Clock (TLC)
0
1
2
MultiPaxos
idle
instance proposing
done
idle
idle
Paxos Proposer
colecting accept
proposing
colecting promises
preparing
idle
Tag()
consensus on the value
Peer
Parsing accept
Collecting accepts
if consensus on a value, broadcast TLC message
Peer
Parsing TLC
Collecting TLC
if enough TLC received, notify multi-paxos and move to next paxos instance

Figure 1: Timing diagram illustrating how the logical clock evolves with multi-paxos, paxos, and the messages exchanged, on a specific peer. Each clock step corresponds to a Paxos instance. In the first step consensus with paxos is reached. In the second step, no proposal has been made but a quorum (threshold) of TLC messages has been received. In all cases the multi-paxos and TLC goes to the next step once a quorum of TLC messages has been received.

In summary: Multi-paxos goes to the next step once the peer’s logical clock (TLC) advances to the next step. A peer moves to the next logical clock once it receives a quorum of TLC messages for the current step. A peer broadcasts a TLC message for the current step once its paxos instance reaches consensus on a value, or receives enough TLC messages for the current step.

Blockchain

Each clock step corresponds to a Paxos consensus on a value. The history of clock steps, along with their respecting values, is saved on a cryptographically linked-list: a blockchain. A blockchain is a list of blocks cryptographically linked together to form a singly linked list. In our case, each block represents a clock step and contains the step number, the hash of the previous block (this is the cryptographic link), the value corresponding to the paxos consensus value, and the block's hash. Since the step number starts from 0 and increments at each step, the step number saved in a block can be seen as the block's index: the first block corresponds to the consensus for step 0, the second block for step 1, etc…

Each time a peer advances its logical clock it must create the corresponding block and store it using a provided blockchain store. Using a blockchain has the advantage of providing an immutable and easy to verify representation of the state held in blocks. In this case the history of consensus values that peers have agreed on. Since each block is linked to the previous one via its hash, changing the content of a block would invalidate all subsequent blocks, because the block's hash would be different. Therefore, only the hashes of the last blocks are needed to compare the content of, say, two blockchains from two different peers.

Your tasks

In the following tasks you are progressively going to extend the Peerster implementation by adding a Paxos acceptor and a Paxos proposer. Then, you will support a full Paxos consensus to name metahashes within the Peerster system, and finally you will implement a multi-Paxos extension, storing the name/metahash associations in a blockchain.

In summary, here is what needs to be done:

- [ ] 0: Think about and define your code's architecture
- [ ] 1: Implement Paxos acceptor
- [ ] 2: Implement Paxos proposer and adapt the Tag function
- [ ] 3: Implement multi-Paxos with TLC

Before you begin: New messages introduced

Before going into the technical details, here we introduce you to the new types of message Peerster is going to use. Those messages are defined in **`types/consensus_def.go`**.

PaxosPrepareMessage

This message is used by a peer as part of the Paxos protocol (phase 1), in a proposer role, to send a `PREPARE` message to an acceptor. It contains a **`Source`** field corresponding to the address of the proposer's peer, and an **`ID`** corresponding to the proposal ID. Furthermore, it contains a **`Step`** value representing the current TLC step identifier (equal to the Paxos instance number), so that messages can be attributed to a given instance of Paxos.

PaxosPromiseMessage

This message is used by a peer as part of the Paxos protocol (phase 1), in an acceptor role, to send a `PROMISE` message to the proposer, in response to a **`PaxosPrepareMessage`**. It contains a **`Step`** and an **`ID`** field corresponding to the **`PaxosPrepareMessage`** it is responding to. Furthermore, if the acceptor has already accepted a value (in Paxos phase 2), the message's **`AcceptedId`** and **`AcceptedValue`** fields will contain the accepted value and corresponding ID.

PaxosProposeMessage

This message is used by a peer as part of the Paxos protocol (phase 2), in a proposer role, to send a `PROPOSE` message to an acceptor. On top of a **`Step`** and **`ID`** fields, it contains a **`Value`** field corresponding to the proposed value for the Paxos consensus.

A `PaxosValue` is the structure representing the proposed value. It contains an identifier `UniqID` that uniquely identifies the proposal, as well as a `Filename` and `Metahash` fields corresponding to the proposed association.
The `UniqID` must be generated with `xid.New().String()`.

PaxosAcceptMessage

This message is used by a peer as part of the Paxos protocol (phase 2), in an acceptor role, to broadcast an `ACCEPT` message, in response to an acceptable `PaxosProposeMessage`. It contains the same fields as the `PaxosProposeMessage`.

TLCMessage

This message is used by a peer to signal that the Threshold Logical Clock is ready to be moved forward and share the corresponding blockchain block (`Block`). The block is encoded as a structure of type `BlockchainBlock`.

Recall that a TLC message is sent by a node when it receives a threshold of `ACCEPT` message for the same value (i.e. a consensus is reached at the Paxos layer) OR when it receives a threshold of TLC message for the current step.

A `BlockchainBlock` is a block on the blockchain. It is characterized by its position (`Index`) in the blockchain, a hash of the previous block (`PrevHash`), the `Value` it contains (namely the `PaxosValue`), and its `Hash`, computed over its index, value, and the hash of the previous block. Use `crypto/sha256` to compute the hash.

Task 0: Think about and define your code's architecture

In this homework, and unlike the preceding ones, you will not have to implement multiple independent functionalities. Instead, you will exclusively rework the `Tag` functionality from HW2. You will modify it in-depth by implementing a combination of protocols and storage that are built on top of each other. The objective is to ensure that the `Tag` function has an effect on the overall Peerster system, as opposed to being local to one peer.

As such, it is important that you take some time to understand what those protocols require (Paxos and TLC), how they work together and how you can design your code so as to implement them incrementally, without having to rewrite all your code from one task to the next. Also make sure to read the resources about Paxos mentioned. Make sure you understand the different layers we introduced and how they interact: blockchain, TLC, multi-paxos, Paxos (instance).

Paxos is a consensus protocol that allows the peers to agree on a single value. It does this task well, but it is not designed for agreeing on multiple values, moving the system forward, or storing the result of such a consensus.

Multi-Paxos is the extension of the consensus protocol to support multiple instances of Paxos. As such it builds upon Paxos. In our case, we are going to implement it with TLC, effectively ensuring that each Paxos instance happens in a distinct logical clock step and that the result of the consensus is propagated as soon as a Paxos consensus is reached.

On top of providing a multi-Paxos implementation, the TLC messaging also contains the block (to be added to the blockchain) that was generated in the current TLC step. The blockchain itself is nothing more than the collection of the blocks received at each TLC step.

Figure 2 illustrates the relationship between the aforementioned elements. Note that it only illustrates a possible way of organizing components and you are free to come up with other solutions. In fact, everything acts as a black box in the testing environment, where the only input is the `Tag` function, and the outputs are the blockchain and name stores.

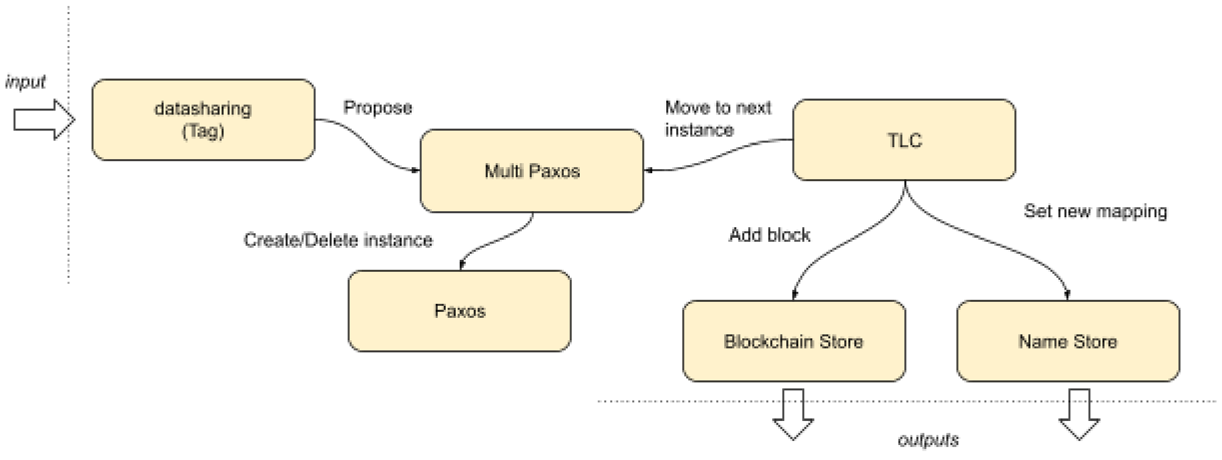


Figure 2: A possible relationship between internal architectural components for HW3. For the testing environment the system is a black box, with the Tag function as input, and the blockchain and name stores as outputs.

Task 1: Implement Paxos acceptor

As a first step in this homework, we suggest you start by implementing a working Paxos acceptor that can support one instance of Paxos. At this step, there's no expectation that your acceptor will be called in more than one instance of consensus. You can thus assume that the `Step` field in Paxos messages will always be 0-valued.

Obviously, you might also want to think ahead and implement the rough design you identified in Task 0 to minimize the amount of code you will have to rewrite later on.

A working Paxos acceptor implementation must be able to respond to both `PaxosPrepareMessage` and `PaxosProposeMessage`.

Upon receiving a `PaxosPrepareMessage`, the peer has to:

1. Ignore messages whose `Step` field do not match your current logical clock (which starts at 0)

2. Ignore messages whose `ID` is not greater than `MaxID`, the previously highest `ID` you have observed within this logical clock step. `MaxID` starts at 0 and a proposer should always use an ID greater than 0.
3. Respond with a `PaxosPromiseMessage` matching the `PaxosPrepareMessage`, including if applicable the value that you have already accepted within this Paxos instance, c.f. the fields `AcceptedValue`, `AcceptedID`. The response must be a broadcast of a private message to the `Source` from the `PaxosPrepareMessage` message.

Upon receiving a `PaxosProposeMessage`, the peer has to:

1. Ignore messages whose `Step` field do not match your current logical clock
2. Ignore messages whose `ID` isn't equal to the highest you have observed within this step (`MaxID`)
3. Broadcast a `PaxosAcceptMessage` matching the `PaxosProposeMessage`.

These rules will ensure that the peer can function effectively as a Paxos acceptor, thereby passing the `Test_HW3_Paxos_Acceptor_` tests.

Task 2: Implement Paxos proposer and adapt `Tag`

Your peer can now work as a Paxos acceptor, it is thus time to extend its functionality to be a working Paxos proposer. The consensus has to be initiated by a Paxos proposer, which is started when calling the `Tag` function. You can extend the function to do just that, while leaving the rest of its implementation as-is for now. We will come back to `Tag` once the consensus is fully implemented and working (Task 3).

Paxos phase 1

In the first phase of the Paxos protocol, the peer, acting as a proposer, needs to broadcast a `PaxosPrepareMessage`, collect the promises, and initiate phase 2 based on the responses it obtained.

When sending the `PaxosPrepareMessage`, you should use a `Step` value of 0 as part of this task, as we don't yet assume any support for multi-Paxos. The `Source` field should match the peer's address and the `ID` value should be the `PaxosID` provided in the configuration.

In response, the acceptors send `PaxosPromiseMessage`'s. Upon receiving them, the peer has to:

1. Ignore messages whose `Step` field does not match your current logical clock.
2. Ignore messages if the proposer is not in Paxos phase 1.
3. Collect the `PaxosPromiseMessage`'s until a threshold of peers replies. The threshold function is provided in the configuration. It takes in input the total number of peers, which is also provided in the configuration, and returns the minimum number of

peers to reach a threshold. Use that function here and elsewhere to compute the threshold.
4. Retry with the next ID if after the provided `PaxosProposerRetry` timeout there are not enough promises collected. Recall that the next ID is the current ID plus the total number of peers.

Once a threshold of peers replies with a `PaxosPromiseMessage` message, the peer, acting as a proposer, initiates phase 2.

Paxos phase 2

The proposer initiates phase 2 of the Paxos protocol by broadcasting a `PaxosProposeMessage` containing:

- the peer's proposed value (metahash/filename association contained in a `PaxosValue`), if no `PaxosPromiseMessage` contained an `AcceptedValue`, or
- the `AcceptedValue` from the `PaxosPromiseMessage` with the highest `AcceptedID`

In response, the acceptors broadcast a `PaxosAcceptMessage`. Upon receiving them, the peer has to:

1. Ignore messages whose `Step` field do not match the peer's current logical clock
2. Ignore messages if the proposer is not in Paxos phase 2
3. Collect the `PaxosAcceptMessage` until it has a threshold of the same value (use `UniqID` to group values).
4. Retry from phase 1 with the next ID if after the provided `PaxosProposerRetry` timeout there is not a threshold number of promises with the same values collected.

Once the threshold has been reached, we know that we have reached a consensus. The Paxos instance can be considered finished and, in a multi-Paxos implementation, we would be ready to advance to the next Paxos instance. You should be able to run the `Test_HW3_Paxos_Proposer_` tests. We are ready to move to Task 3 !

Task 3: Implement Multi-Paxos with TLC

Your peer can now work as a Paxos acceptor and proposer, the Peerster system can reach a single consensus and we are ready to update the `Tag` function that was developed in HW2. At the end of this task, all tests should pass.

TLC

As the Paxos functionality is ready, you can now extend it to a full multi-Paxos implementation based on TLC. In concrete terms, this means that when a peer observes that the consensus has been reached, it will broadcast a `TLCMessage` and wait for a threshold of `TLCMessage` to advance their TLC step. Recall that a peer knows a consensus has been reached for the current step once it observes a threshold of `PaxosAcceptMessage` for the same value that references the current step in their `Step` field.

When the consensus is observed, you must build the blockchain block that corresponds to the step before broadcasting the `TLCMessage`. The block's `Index` is the current step number. The block `Value` can be taken from one of the `PaxosAcceptMessage`. The previous hash (`Prevhash`) is the hash of the previous block (use the blockchain store to get it). In case this is the first block it must be a slice of 32 zero-bytes. Finally, you must compute the `Hash` of the block as follow:

```
H = sha256(Index || v.UniqID || v.Filename || v.Metahash || Prevhash)
```

The Index must be converted using `strconv.Itoa` and strings can be transformed with `[]byte(<string>)`. "v" represents the `PaxosValue`. Use `crypto/sha256` to compute the hash.

Once the block has been computed, the peer can broadcast its `TLCMessage` containing the block and current step.

When receiving a threshold of `TLCMessage`'s for its current step, the peer should:

1. Add the block to its own blockchain (see next section)
2. Set the name/metahash association in the name store
3. In case the peer hasn't broadcasted a `TLCMessage` before: broadcast the `TLCMessage`
4. Increase by 1 its internal TLC step
5. Catchup up if necessary: Loop to point 1 in case it has already collected enough TLC messages for the new step, BUT skip the point 3. A peer doesn't need to broadcast its TLC message when it does a catchup

As stated earlier, `TLCMessage`'s concerning past steps can be ignored, whereas those concerning future steps should be stored for further processing.

Blockchain

To store the blocks, we require you to use a `storage.Store`. Use `GetBlockchainStore()` from the `Storage` in the configuration. We ask that you store every block indexed by its hex-encoded hash value. As the store only handles bytes, you need to marshal/unmarshal blocks. For that you must use the `blockchain.Marshal` and `blockchain.Unmarshal` functions. Listing 1 illustrates how to use those functions.

```
// Marshal
buf, err = block.Marshal()
if err != nil {...}

// Unmarshal
var block types.BlockchainBlock
err := block.Unmarshal(buf)
if err != nil {...}
```

Listing 1: Example of Marshal / Unmarshal for a blockchain block

Additionally, the last block's hash (not hex encoded) must be stored with the key `storage.LastBlockKey`. Update that key's content each time a new block is added. This makes it possible to navigate through the blockchain by accessing its last block, which contains the hash of the previous block, which indexes the previous block, and so on recursively.

Finally, update the Tag function

The `Tag` function must use the consensus mechanism when the `TotalPeers` parameter from the config is greater than 1. If that's the case, the Tag function must return an error if the name already exists in the name store. Otherwise, it must return only once the value has been accepted by the consensus, i.e. a block has been added with the proposed value on a quorum of peers. Figure 3 illustrates the activity when the Tag function is called. Note that it does not illustrate the TLC and blockchain parts.

Only the name must be unique: once a name is used it can't be used anymore. But a metahash can have multiple names associated with it.

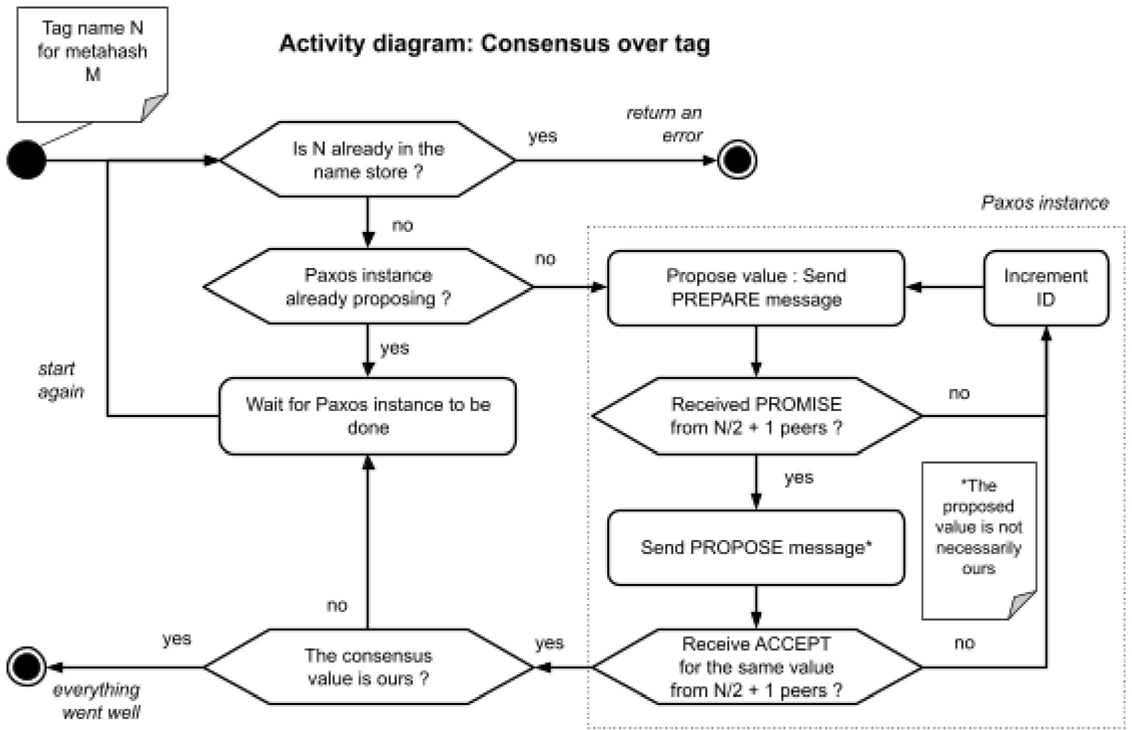
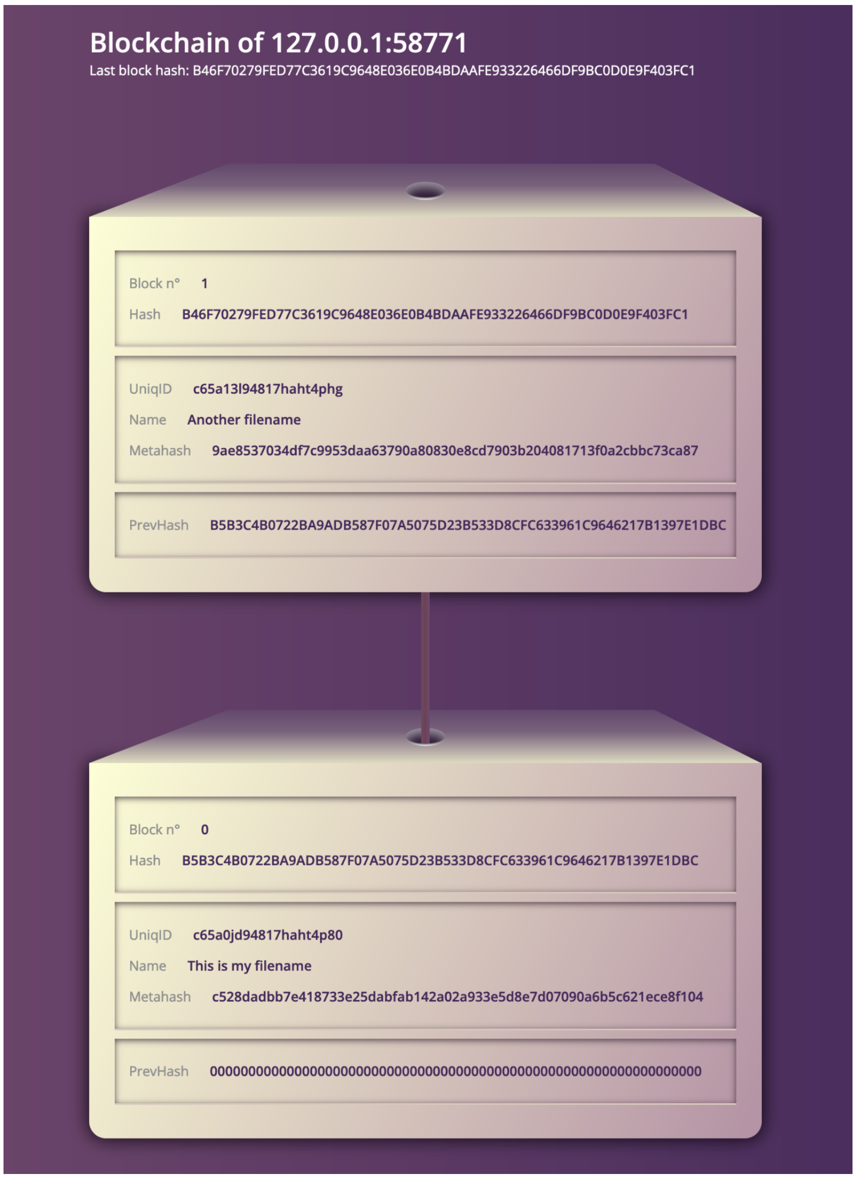

Tag name N for metahash M
Activity diagram: Consensus over tag
Is N already in the name store ?
yes
return an error
no
Paxos instance
Paxos instance already proposing ?
no
Propose value : Send PREPARE message
Increment ID
yes
start again
Wait for Paxos instance to be done
Received PROMISE from N/2 + 1 peers ?
no
yes
Send PROPOSE message*
*The proposed value is not necessarily ours
no
yes
The consensus value is ours ?
yes
Receive ACCEPT for the same value from N/2 + 1 peers ?
no
everything went well

Figure 3: Activity diagram when the Tag function is called and TotalPeers is greater than 1. It does not illustrate the TLC and blockchain part, which would happen when the activity terminates ("everything went well"). TLC is what would terminate the "Wait for Paxos instance to be done" job. A paxos instance is done once the TLC clock moves to the next step.

Try your program

As in HW2 and apart from the unit tests and integration tests, you can use the web GUI to try your program. Use the `--totalpeers` and `--paxosid` CLI arguments to make use of the consensus on the Tag functionality. When a tag is performed on a peer you should eventually be able to resolve the name on another peer.

You can visualize a peer's blockchain by visiting the `/blockchain` endpoint when a peer is running via the CLI. You should see something as illustrated with figure 4.



Blockchain of 127.0.0.1:58771

Last block hash: B46F70279FED77C3619C9648E036E0B4BDAAFE933226466DF9BC0D0E9F403FC1

Block n° 1

Hash B46F70279FED77C3619C9648E036E0B4BDAAFE933226466DF9BC0D0E9F403FC1

UniqID c65a13l94817haht4phg

Name Another filename

Metahash 9ae8537034df7c9953daa63790a80830e8cd7903b204081713f0a2cbbc73ca87

PrevHash B5B3C4B0722BA9ADB587F07A5075D23B533D8CFC633961C9646217B1397E1DBC

Block n° 0

Hash B5B3C4B0722BA9ADB587F07A5075D23B533D8CFC633961C9646217B1397E1DBC

UniqID c65a0jd94817haht4p80

Name This is my filename

Metahash c528dadbb7e418733e25dabfab142a02a933e5d8e7d07090a6b5c621ece8f104

PrevHash 0000000000000000000000000000000000000000000000000000000000000000

Figure 4: Illustration of the page showing the blockchain of a peer. This page is accessible via the `http://<peer proxy address>/blockchain` *URL when a Peer is running. Note that you must reload the page to see changes.*

Appendix A - Tests definitions

Tag with no consensus

Pre-condition	**Action**	**Expectation**
(3-1) P1 started with TotalPeers set to 1 P2 started and TotalPeers set to 0	Call Tag on P1 Call Tag on P2	P1 send no message P2 send no message

Paxos acceptor prepare

Pre-condition	**Action**	**Expectation**
(3-2) A peer “Acceptor” running Socket “Proposer” running The Acceptor knows the Proposer	Proposer sends a Paxos prepare to the Acceptor with step=99 and ID=1	The acceptor ignored the message (wrong step): no messages sent, block store is empty, name store is empty.
(3-3) A peer “Acceptor” running Socket “Proposer” running The Acceptor knows the Proposer	Proposer sends a Paxos prepare to the Acceptor with step=0 and ID=0	The acceptor ignored the message because the ID is too low: no messages sent, block store is empty, name store is empty. (The ID of a proposer is always >0)

(3-4) A peer “Acceptor” running Socket “Proposer” running The Acceptor knows the Proposer	Proposer sends a Paxos prepare to the Acceptor with step=0 and ID=99	The acceptor sends back the corresponding promise.

Paxos acceptor propose

Pre-condition	**Action**	**Expectation**
(3-5) A peer “Acceptor” running Socket “Proposer” running The Acceptor knows the Proposer	Proposer sends a Paxos propose to the Acceptor with step=99 and ID=1	The acceptor ignored the message (wrong step): no messages sent, block store is empty, name store is empty.
(3-6) A peer “Acceptor” running Socket “Proposer” running The Acceptor knows the Proposer	Proposer sends a Paxos propose to the Acceptor with step=0 and ID=2	The acceptor ignored the message (wrong ID, 0 is expected): no messages sent, block store is empty, name store is empty. MaxID = 0
(3-7) A peer “Acceptor” running Socket “Proposer” running The Acceptor knows the Proposer	Proposer sends a Paxos prepare to the Acceptor with step=0 and ID=5 Proposer sends a Paxos propose to the Acceptor with step=0, ID=5 and Value=V Proposer sends a Paxos prepare to the Acceptor with step=0 and ID=9	The acceptor sent back a private paxos promise to the proposer that contains AcceptedValue=V and AcceptedID=5

(3-8) A peer “Acceptor” running with a threshold of 1. Socket “Proposer” running The Acceptor knows the Proposer	Proposer sends a Paxos propose to the Acceptor with step=0 and ID=0.	The acceptor broadcast an Accept in a rumor.

Paxos proposer

Pre-condition	**Action**	**Expectation**
(3-9) A peer “Proposer” running provided with PaxosID P Socket “Acceptor” running The Proposer knows the Acceptor	Call Tag() on the proposer in a go routine. The acceptor send back a promise with Step=99 and the ID=P.	The acceptor receives a prepare. The proposer sent two messages: the prepare broadcast, and the promise from itself to itself. The proposer ignores the promise from the Acceptor: still sent only 2 messages in total, block store is empty, name store is empty.
(3-10) A peer “Proposer” running with a threshold of 2 and PaxosID P Socket “Acceptor” running The Proposer knows the Acceptor	Call Tag() on the proposer in a go routine. The acceptor sends back a correct promise with Step=0 and ID=P.	The acceptor receives a prepare. The proposer broadcasts a corresponding correct propose message.

TLC

Pre-condition	**Action**	**Expectation**
(3-11) P1 running with a threshold of 2. Socket P2 running P1 knows P2	P2 sends a correct TLC message to P1 with Step=0. P2 sends a TLC message to P1 with Step=1.	P1 has nothing in its block store (it didn't advance its TLC clock) (only received 1 TLC message for step 0)
(3-12) P1 running with a threshold of 2. Socket P2 running P1 knows P2	P2 sends 2 TLC messages to P1 with Step=0.	P1 has one block in its block store. P1 has the block's hash in the LastBlockKey block store. P1 has the corresponding name=>metahash in its name store.
(3-13) P1 running with a threshold of 1. Socket P2 running P1 knows P2	P2 sends a TLC message to P1 with Step=2. P2 sends a TLC message to P1 with Step=1. P2 sends a TLC message to P1 with Step=0.	P1 has no blocks in its block store. P1 has 3 blocks in its block store with the correct blocks. P1 has the last block's hash in the LastBlockKey block store.

Simple full scenario

Pre-condition	Action	Expectation
(3-14) The following topology: P1 -> P2	Call Tag() on P1	P1 sends the following rumors, in order: - Rumor(1):PaxosPrepare - Rumor(2):Private:PaxosPromise - Rumor(3):PaxosPropose - Rumor(4):PaxosAccept - Rumor(5):TLC P2 sends the following rumors, in order: - Rumor(1):Private:PaxosPromise - Rumor(2):PaxosAccept - Rumor(3):TLC P1 and P2 have the corresponding name=>metahash in their name stores P1 and P2 have the correct block and last block hash in their block store.

Tag

Pre-condition	Action	Expectation
(3-15) The following topology: P1 <-> P2 The threshold is set to 3. The Paxos retry is set to 4 seconds.	Call Tag() on P1 in a go routine Wait 6 seconds	The Tag() call must be blocking (no consensus reached) P1 retries once and sends the following rumors, in order: - Rumor(1):PaxosPrepare - Rumor(2):Private:PaxosPromise (from P1 to P1) - Rumor(3):PaxosPrepare (second attempt) - Rumor(4):Private:PaxosPromise (from P1 to P1)
(3-16) The following topology: P1 <-> P2 The threshold is set to 3. The Paxos retry is set to 2 seconds. Anti-entropy is set to 1 second.	Call Tag() on P1 in a go routine Wait 3 seconds Add P3 with Heartbeat of 1 hour. P3 knows P1. Wait 10 seconds	The Tag() call must be blocking (no consensus reached) A consensus is reached. Name stores in P1, P2, P3 contain the corresponding name=>metahash P1, P2, P3 broadcast a TLC message.

(3-17) P1, not started. Name store or P1 contains the association N=>M	Call Tag() on P1 for N=>M	Tag() returns an error.
(3-18) The following topology, with the default threshold. P1 <-> P2	Call Tag() on P1 T times Add P3. P3 knows P2 P3 broadcast an empty message	P1 and P2 have T blocks in their block stores and T name=>metahash associations in their name store. P3 has T blocks in its block store and T name=>metahash associations in its name store. Blockchains in P1, P2, P3 are valid. Last blocks of P1, P2, P3 have the same hash.
(3-19) K peers running and all knowing each other.	Make every K peer call Tag() asynchronously T times. Wait on all peer to be done calling Tag()	All peers have a valid blockchain with K*T blocks and the same LastBlockKey All peers have K*T elements in their name store.

Integration test

Same as (3-19) but with a mix of reference and student peers.