Implementing OmniLedger sharding

Quang-Minh Nguyen

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Semester Project

December 2018

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Linus Gasser
EPFL / DEDIS

Contents

1 Introduction

The recent success and surge of popularity enjoyed by Bitcoin has sparked the curiosity of many. Beyond the economical aspects of the cryptocurrency, lies an intriguing distributed ledger technology capable of processing transactions in a decentralised manner without compromising on security. Traditionally, transaction processing systems were designed with a centralised philosophy in mind: a transaction between two parties is processed by a third party. One of the main problem of this design is the fact that this third party constitutes a single point of failure in the system, which makes it an attractive target for security attacks. In Bitcoin, there is no such central entity. Instead, a large number of nodes participating in the network, called miners, validate transactions concurrently and come to an agreement via the use of Nakamoto's consensus. For these reasons, among others, some consider Bitcoin to offer the first successful decentralised transaction processing system. Consequently, blockchain technology has gained a lot of interest in recent years. There has been an increasing amount of research on finding the best possible applications for this innovative technology but also new distributed ledger designs aiming to provide interesting features or attempt to correct existing problems.

One of the key challenges blockchain technology faces today is to offer performance on par with more traditional centralised payment processors (e.g. Visa). In particular, how to achieve scalability, i.e. increase the total transaction volume with the number of participants. Many approaches have been considered, however most of them trade either security or decentralisation for scalability. The next section briefly introduces OmniLedger[14], a novel scale-out distributed ledger that preserves long-term security under permissionless operation.

1.1 OmniLedger

OmniLedger is a permissionless distributed ledger capable of scaling-out while preserving security. It is designed to achieve higher throughput and shorter confirmation time via sharding. The idea of sharding is to divide the ledger state into multiple parts (shards) and assign to each a different subset of validators. For example, in an account-based ledger, all transactions involving accounts from Europe would be handled by shard A, while transactions involving accounts from Asia would be handled by shard B. This partitioning will allow each shard to process transactions in parallel. As a result, the individual transaction processing load will be reduced as a validator will only process transactions given to its shard, instead of every transaction. Moreover, the system processing power will increase proportionally with the number of validators as the addition of new participating

nodes will results in more shards.

The main goal is to achieve sharding in a decentralised and secure manner while providing good performance. Here are described briefly a few of the most important challenges in achieving that goal and how OmniLedger overcomes them. First, the sharding assignment must be bias-resistant, the design employs distributed randomness generation protocols (e.g. RandHound[15]) which generate verifiable randomness in a unbiased manner. Such randomness can then be used to generate shard assignments. Next, there needs to be a way to process transactions involving different shards, in particular how to make sure such transactions are either committed or aborted atomically. To that end, OmniLedger introduces Atomix, a two-step cross-shard transactions commit protocol where the idea is to first send the transaction to every input shard, then collect their individual answers, either an acknowledgement or an error. Then, abort if there is an error, otherwise commit the transaction to the output shards. The last challenge is to offer better support for low-value payments. Currently, transactions in blockchain technology systems have high-latency regardless of their value due to eventual consistency (e.g. Bitcoin recommends waiting for six subsequent blocks before considering a transaction as permanently on the blockchain). This is a problem especially for low-value transactions where the stakes are not high enough to warrant such a long wait. The solution OmniLedger proposes is to use a two-tier validation scheme called Trust-but-Verify. In this scheme, transactions are sent from the client to a first tier of validators called optimistic validators, then to a second tier called core validators before finally being added to a block. The difference between the two tiers lies size: optimistic validators form smaller shards than core validators, this allows them to process transactions faster at the expense of some security. Thus, low-payment transactions can already be accepted after the first validation, while seller can choose to wait for both validation steps for higher-stake transactions.

An implementation of OmniLedger is currently under development in the Decentralized and Distributed Systems lab (DEDIS) at EPFL[5]. In its current state, the project supports many features including smart contracts, creation of individual ledgers, multiple transactions per block. However it does not include sharding nor integrate multiple ledgers.

The goal of this project is to incorporate a first sharding algorithm to the current OmniLedger project. Implementing sharding and all the necessary mechanisms to make it secure and decentralised (as described previously) takes more time than a semester project allows. Thus, this project focuses more on creating a solid foundation, such that future works can continue to build upon it, towards a complete implementation. A well-written, easy

to understand implementation, in addition to documentation and testing, is required in order for the code to be improved and refactored in subsequent work.

2 Architecture

This section gives an overall view of the architecture and describes briefly the role of each component. A diagram summarising the architecture can be found in figure 1

2.1 Cothority

The Omniledger project is part of the collective authority or Cothority project[6]. Cothority is a framework for development, analysis and deployment of decentralised, distributed protocols. It is maintained and developed by the DEDIS lab at EPFL. The project's code is open source. Most Cothority projects are written in Java or Go, OmniLedger is written in the latter.

In Cothority, protocols are run on a given set of servers, called cothority servers or conodes, while the whole is referred to as a collective authority or cothority. In its current state, the project contains a few applications, such as ByzCoin, Onchain-Secrets and Proof of Personhood.

This project builds on top of ByzCoin, which already implements individual ledger with multiple transactions, a consensus protocol and smart contracts among other functionalities. In addition, the project also employs the ONet[4] and protobuf[9] packages. ONet or Overlay-network is the Cothority Network Library, it provides network abstraction functionalities to facilitate the deployment and test of decentralised protocols. Protobuf offers reflection-based protocol buffers, it allows the user to define message formats, encode and decode Go data structures. It is mainly used in inter-node communication.

2.2 OmniLedger architecture

In terms of architecture, an OmniLedger can be seen as a collection of ledgers composed of an identity ledger and several shard ledgers. To communicate with an OmniLedger, several components are required: A front-end client, an OmniLedger client and service, a ByzCoin client and service. In addition, smart contracts are also part of the architecture, as they define methods that can be executed on the ledgers.

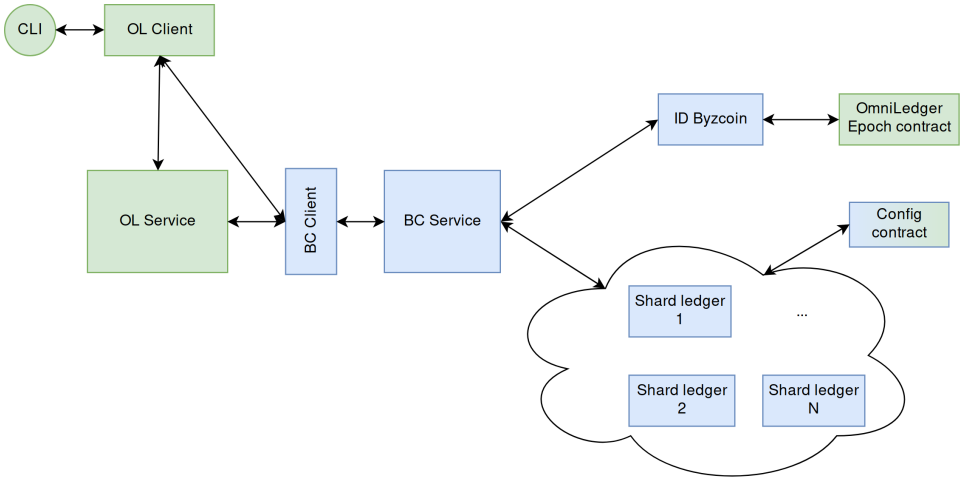

Figure 1: The OmniLedger architecture. Elements in green were implemented in this project. Elements in blue were already part of the Cothority framework. Config contract is in both colours as some code was added to it. The cloud of shard ledgers is used to abstract communication for every shard ledger, it is meant to imply that the BC service communicates with each shard ledger individually for example.

2.2.1 Identity Ledger & Shard ledgers

The identity ledger, also referred to as identity ByzCoin or IB, has two responsibilities. The first is to keep track of nodes participating in the OmniLedger (often referred to as IB roster), as well as the roster of each shards. These two pieces of information, in addition to other configuration parameters (shard count, epoch size) are stored in the ledger state. The second is to compute the sharding assignment when the OmniLedger is created and in every subsequent new epochs.

A shard ledger is responsible for processing transactions. It also holds configuration information such as its roster, the maximum block size and the block interval.

Both the IB and the shard ledgers are implemented as ByzCoin ledgers. As a result, ByzCoin clients and the ByzCoin service must be used to communicate with them.

2.2.2 CLI client

The CLI (command-line interface) client serves as frontend for OmniLedger. It does some parameter checking before generating a request and sending it to OmniLedger service via the OmniLedger client. It was written with the

help of the cli[1] package. The CLI is accessible via the `oladmin` executable.

At this moment, the following commands are available:

- `oladmin create -shards <shard_count> -epoch <epoch_size> <roster_file>`
  Creates a new OmniLedger with `<shard_count>` shards and `<epoch_size>` epoch size. The command will generate two local `.cfg` files. The first contains the configuration of the created OmniLedger (roster of conodes, IB identity, shard identities, admin identity, etc). The second contains the ledger admin's keys.
  - `<shard_count>` should be low enough to form sufficiently large shards.
  - `<epoch_size>` is in milliseconds and indicates the minimum amount of time between two epochs.
  - `<roster_file>` is a `.toml` file containing the address, public key and a description of each participating conode.
- `oladmin newepoch <omniledger_cfg> <key_cfg>`
  Requests a new epoch for the OmniLedger specified by `<omniledeger_cfg>`.
  - `<omniledger_cfg>` file holding the targeted OmniLedger configuration.
  - `<key_cfg>` file holding the keys of the admin of the targeted OmniLedger.
- `oladmin status <omniledger_cfg>`
  Prints the current OmniLedger roster and the shard rosters.
  - `<omniledger_cfg>` file holding the targeted OmniLedger configuration.

2.2.3 OmniLedger Client & Service

The OmniLedger service acts as an intermediate between the client and an OmniLedger. Its main role is to transform user requests into transactions to be sent to the ledgers. Its responsibilities include:

- Defining accepted user requests, their message structures and registering them.
- Processing user requests: checking parameter correctness, creating and sending the corresponding transactions, return response to the client.
- Coordinating the IB and the shard ledgers.
- Registering smart contracts.

2.2.4 OmniLedger client

The OmniLedger client is used to access with the OmniLedger service. It defines the API available from the front-end client. Even though in theory it should only forward requests to the OmniLedger service, in practice the client sometimes contact the ledgers directly.

2.2.5 ByzCoin Client & Service

The ByzCoin service is used to communicate with the different shards composing an OmniLedger. It is part of the ByzCoin project, and thus was only used as a black-box for sending requests to the IB or the shard ledgers. Similarly to the OmniLedger service, the ByzCoin service is accessed by using a ByzCoin client.

2.2.6 Smart Contract and Instances

OmniLedger and ByzCoin implement smart contracts. In short, they allow the definition of structures to be stored on the blockchain, as well as the definition of executable methods on these structures (more details on smart contracts and their implementation can be found in [2]). In this project, a new contract type was implemented: the OmniLedger Epoch contract. In addition, a new instruction was added to the Config contract of ByzCoin.

The OmniLedger Epoch contract stores an OmniLedger configuration as instance data, it contains the roster, the shard count, the epoch size, a timestamp and the shard rosters. The contract supports two instructions. The first is `spawn:omniledgerepoch`, it is used to set up a new OmniLedger. The second instruction is `invoke:request_new_epoch`, which is used to request a new epoch. More details on both instructions and how they are used in their respective context can be found in section 3.4 and 3.5.

The Config contract of Byzcoin was extended to support a new `invoke:new_epoch` instruction. The execution of this instruction will compute the new shard roster and update the configuration.

3 Implementation

This section explains the different protocols and algorithms OmniLedger uses, as well as how they were implemented.

3.1 Sharding assignment

The sharding assignment algorithm takes as input a roster R (i.e. list of nodes), a number of shard n and an initial seed s. It starts by computing

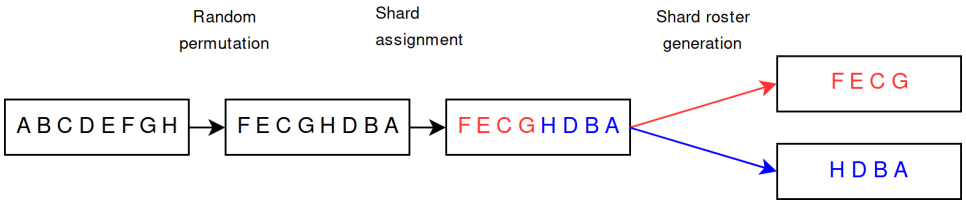

Figure 2: Example of a sharding assignment execution

a random permutation of the nodes in R using the given seed s. Then, it forms n shards based on the previously computed random permutation. The algorithm is designed to compute a "fair" sharding assignment, i.e. the difference in size between any two shards is at most one. The algorithm has two main steps, the first is to slice the roster such that every shard has the same number of nodes, the second is to assign the remaining nodes uniformly.

R should be the identity ledger roster, i.e. all nodes participating in the network. n should be low enough such that the resulting shards have enough nodes to ensure some security properties: Omniledger employs ByzCoinX[13] as its consensus protocol, it provides consensus in a Byzantine setting as long as the number of malicious nodes is less than a third of the total number of validators.

A new sharding assignment is computed during an OmniLedger's creation and at the start of a new epoch.

As for the implementation, the sharding assignment is computed by a function, which is called in the OmniLedger Epoch contract when executing the `spawn:omniledgerepoch` or the `invoke:req_new_epoch` instruction. The implementation currently uses the hash of the received instruction as seed. This is not secure as an attacker could change the seed by modifying the instruction (instructions are signed with a Schnorr signature which employs a random component during the signature generation, as a result an attacker could sign the instruction multiple times until it obtains a seed to their liking). The solution would be to use a bias-resistant distributed randomness protocol, e.g. RandHound[15], as mentioned in the OmniLedger paper.

3.2 Applying roster change

Once a new sharding assignment has been computed, the changes must be communicated and applied to the shards. New shard rosters are applied shard-by-shard. To go from the current shard roster R to the new shard roster R', the algorithm always start by adding new nodes (i.e. *nodes* such that $nodes \in R' \wedge nodes \notin R$) and then removes old nodes (i.e. *nodes* such

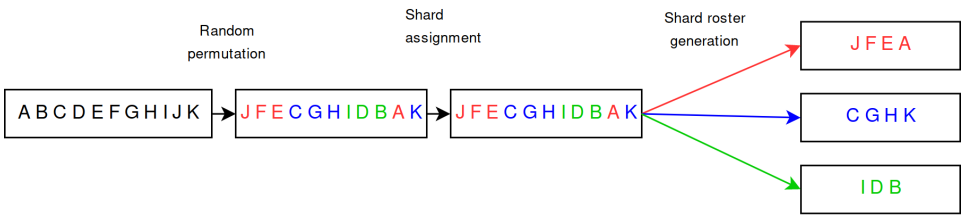

Figure 3: Another example of a sharding assignment execution where the concept of "fair" assignment is illustrated: nodes A and K are assigned to roster 1 and 2 respectively, instead of both being assigned to roster 1

that $nodes \notin R' \wedge nodes \in R$). One critical property of the algorithm is that it is designed to add or remove nodes one-by-one instead of in batches. This is done to ensure that shards will always have enough validators to keep processing transactions during the transition from old to new roster.

In terms of implementation, the application of roster change is part of the Config smart contract of ByzCoin, in particular, one can execute it by sending a transaction containing the `invoke:new_epoch` instruction. The latter must contain as arguments a proof of the new epoch (obtained from requesting a new epoch from the IB) and the index of the shard to be updated. Upon receiving the instruction, the shard ledger starts by verifying the proof, if it's valid it decodes it to get the new roster. Then, it will fetch the current roster from the instance data. Finally, it computes the intermediate roster using the change roster function and update the current roster to the intermediate. Note that because changes are applied one at the time, the same transaction must be sent multiple times to the ledger in order to completely change the shard roster from old to new. This implies the shard roster, and consequently also the instance data and the ledger state, will change gradually.

The examples below show how roster changes are computed. Notice that after the second call in the first example, the order in the returned roster changed, in particular new nodes were put in front of the roster. The algorithm puts new nodes in fron of the roster when all nodes have been added. The motivation behind such a change is to guarantee that the leader won't be removed (in ByzCoin, the first node in the roster list is considered the leader). Also notice that the examples employs two shards which do not meet the required number of validators (each should have at least 4 nodes) for the sake of brevity.

```
cur = [A,B];
new = [C,D];
1st call: cur = ChangeRoster(curr, new); // cur is [A,B,C]
2nd call: cur = ChangeRoster(curr, new); // cur is [C,D,A,B]
```

```
3rd call: cur = ChangeRoster(curr, new); // cur is [C,D,B]
4th call: cur = ChangeRoster(curr, new); // cur is [C,D]


cur = [A,B,C];
new = [C,D,E];
1st call: cur = ChangeRoster(curr, new); // cur is [A,B,C,D]
2nd call: cur = ChangeRoster(curr, new); // cur is [C,D,E,A,B]
3rd call: cur = ChangeRoster(curr, new); // cur is [C,D,E,B]
4th call: cur = ChangeRoster(curr, new); // cur is [C,D,E]
```

3.3 Timestamps

Timestamps are used in both the creation of a new OmniLedger and the request of a new epoch. The need for timestamps stems from one of the requirement in the new epoch protocol: to check if enough time has elapsed since the previous successful new epoch. Thus, the idea was to add a timestamp, computed on the client side, to user requests, and use it to determine the validity of the request, i.e. a new epoch request is valid if the difference in time between its timestamp t_{req} an the previous new epoch's timestamp t_{prev} is larger than the epoch size es or $|t_{req} - t_{prev}| \geq es$. Consequently, request for the creation of a new OmniLedger also had to include a timestamp to mark the first successful epoch.

However, this came with some security issues: An attacker could manipulate t_{req} by changing the internal clock of its machine before the client program computes it. To mitigate this problem, the OmniLedger Epoch contract includes an additional check: it verifies that t_{req} is within some time window from the node's clock nc, i.e. t_{req} is at most either w seconds earlier or later than the node's clock for the request to be considered valid or $|t_{req} - nc| \leq w$. The value of w is a parameter that is hard-coded at the moment (current value is 60 seconds), but one could refactor the code to make it an OmniLedger parameter.

3.4 OmniLedger creation

The OmniLedger creation protocol is used to set up an new OmniLedger. It is composed of multiple steps from which include the creation of the IB, the initial sharding assignment, the creation of the shard ledgers and so on. Below is a detailed description of each steps.

1. The protocol starts with the execution of the `create` command from the CLI client, it must provide as argument an initial list of participating nodes in a roster file, the number of shards and the epoch size. The client will construct and fill a request partially before sending it

to the OmniLedger client. In particular, it computes and includes a timestamp in the request.

2. The OmniLedger client takes care of creating the missing elements and filling the request before sending it to the OmniLedger service. The missing elements are: a new identity for the owner of the OmniLedger (contains the private/public key pair), a genesis message for the IB and the `spawn:omniledgerepoch` instruction. The latter will not be included in the request directly, instead a transaction containing the instruction will be created and added to the request. The transaction also need to be signed with the private key of the owner. The motivation for doing this operation on the OmniLedger client rather than on the service is to avoid sending the private key over the network, which is unsecure.

3. Upon reception of the request message, the service starts with the creation of the IB ledger. This is done by contacting the ByzCoin service with the previously mentioned genesis message. Next, the service sends the transaction containing the `spawn:omniledgerepoch` instruction to the newly created IB.

4. As the IB receives the transaction, it first check that the request timestamp is not too different from the node's clock (see section 3.3). Then, it computes sharding assignment using the roster provided in the transaction before saving the roster, the shard count, the epoch size, the timestamp and the shard rosters as instance data and return state changes. This whole step is part of the OmniLedger Epoch contract.

5. Once the transaction is on the IB, the service can fetch the newly computed shard rosters. To do this, the service asks the IB for a proof that the spawn instruction was well executed. After verifying the proof, the service can decode it to get the instance data saved in the previous step, among which are the shard rosters. Now, the service can create the shard ledgers similarly to how the IB was created: for each shard, a genesis message contaning the corresponding shard roster is created, before being sent to the ByzCoin service to create the ledger.

6. After the creation of the shard ledgers, the service can send a response to the CLI client. It includes the identity ledger's ID and roster, the shard ledgers' IDs and the OmniLedger Epoch instance ID. In addition, the key pair generated in the earlier steps is also returned to the CLI client.

7. When the client receives the response, it saves the relevant information in a config file and the generated key pair in a key file.

The appendix contains a diagram annotated with step numbers showing communication flows during the execution of the protocol.

3.5 New epoch

The new epoch protocol is used to request the start of a new epoch. It begins with the computation of a new sharding assignment before communicating the new rosters to the shard ledgers. A detailed description of each steps can be found below.

1. The protocol starts with the execution of the `newepoch` command from the CLI. The user must provide the configuration file of the Omniledger it desires to speak to and the corresponding key file. The client will load both files and build a request. Once again, a timestamp is added before the request is sent to the service via an OmniLedger client.

2. The OmniLedger client starts by preparing a transaction containing a `invoke:request_new_epoch` (signing the transaction, incrementing the signer counter) and sends it to the OmniLedger service.

3. The OmniLedger service forwards the transaction to the IB ledger.

4. The IB executes the code according to the OmniLedger Epoch contract: it starts by comparing the timestamp of the request with the node's timestamp to make sure they're not too different. Next, it fetches the instance data to get the current roster, as well as the epoch size and the timestamp marking the previous successful new epoch. Then, it computes the difference between between the request timestamp and the instance data timestamp and checks that it is larger than the epoch size. If this is the case, the IB can compute a new sharding assignment using the instruction's hash as seed, update the instance data with the new timestamp and roster before finally returning state changes.

5. Once the new sharding assignment is completed, the OmniLedger service fetches the proof of the instruction and sends it back to the OmniLedger client as response.

6. The OmniLedger client verifies and decodes the proof to get the new shard rosters. For each shard, the client creates and prepare a transaction with a `invoke:new_epoch` instruction which contains the new roster. As described in section 3.2, this transaction must be sent multiple times (with a few changes such as incrementing the signer counter and re-signing) to the shard ledger until all changes are applied. The exact number of changes can be computed as the set difference between the old and new roster of the shard, i.e. the number of nodes present in the old roster but not in the new one.

7. Once all roster are applied, the OmniLedger client can return the IB roster to the CLI client. The latter will save it in the configuration file.

The appendix contains a diagram annotated with step numbers showing communication flows during the execution of the protocol.

3.6 Get status

The get status protocol is used to learn about the current rosters (IB and shard) of an OmniLedger. Its steps are described below.

1. The protocol starts with the execution of the `status` command from the CLI. The command takes as parameter the configuration file of the OmniLedger the user desires to speak with. The configuration file holds, among other things, the identity of the OmniLedger's IB and the identity of the instance containing the rosters information. The CLI client will create and fill the request, then send it to the OmniLedger client.

2. The OmniLedger client sends the request directly to the ByzCoin service via a Byzcoin client.

3. The OmniLedger client obtains a reponse containing a proof to verify and decode from the BC service.

4. After decoding the proof, it creates and fills a response message with the current IB roster and shard rosters, then sends it to the CLI client.

5. The CLI client can output the current roster and shard rosters.

The appendix contains a diagram annotated with step numbers showing communication flows during the execution of the protocol.

3.7 Testing

In addition to the implementation code, automatic tests were written. There are two types of test in this project: Unit tests and integration tests.

Unit tests were written in Go using a mix of the built-in package `testing`[11], ONet's local testing framework[4], and the `Testify`[10] library. They test the more important functions such as new roster computation or the application of roster changes. To run unit tests, execute the Go command `go test` in the relevant directory. An individual test can also be run by using the command `go test -run <Name of test>`.

Integration tests are written as shell scripts and use ONet's specialised testing bash-library[3]. They aim to validate the integration and interaction of the different components, from the CLI-client call to the contract execution. In particular, tests were written for new Omniledger creation, new epoch request and status request. To run the integration tests, execute the shell script `omniledger/oladmin/test.sh`. In its current state, shard assignment has only been tested on a local machine with 8 conodes.

3.8 Documentation

In order to facilitate code understanding and code re-use, public functions and structures were documented with comments. This is also the case for important private functions.

4 Challenges and Limitations

There were three main challenges during this project. The first challenge was to get familiar with the code present in the Cothority framework, as well as other libraries written by DEDIS (e.g. ONet), in order to use them correctly in the project. In particular, understanding the global structure (client, service, contracts, etc) and the intricacies related to writing a decentralised, secure application.

The second challenge was to regularly follow code changes made by the engineering team and re-factor the Omniledger code to be compliant with these changes.

The third and last challenge was debugging. Due to the distributed nature of the application and the intrinsic high-coupling of this project with the rest of the framework, it was sometimes difficult to find bugs, debug them and test the code in an efficient manner without exterior help.

Due to time constraints and the fact that these challenges slowed down progress, the original scope of the project had to be slighty reduced and a few shortcuts were taken. As a result, the sharding feature may seem basic in its current state, some components described in the Omniledger paper are not implemented yet (e.g. cross-shard transactions or distributed randomness generation). Moreover, performance has only been tested with a small numbers of shards on a local machine.

5 Future work

- Performance tests: As mentioned in section 4, the implementation has only been tested on a small number of nodes. Further testing is required to assess performances.

- Secure sharding: As mentioned in section 3.1, the current sharding algorithm implementation derives a seed from the instruction which requests a new epoch, which is not secured. In the Omniledger paper, distributed randomness protocols, such as RandHound[15] or Drand[7], are used to generate the seed. They provide a way to generate verifiable randomness in a distributed setting, while also ensuring some security properties (e.g. bias-resistance).

- Cross-sharding transaction: The current implementation does not feature cross-shard transaction processing. An improvement would be to implement Atomix[14], a Byzantine Shard Atomic Commit protocol. Atomix employs a lock-then-unlock process to atomically handle transactions across shards. The protocol starts by sending the transaction to every input shard and waits for their respective reply (accept or reject). If any shard rejects the transaction, then the input can be reclaimed, otherwise the transaction is committed to the output shards.

- Optimisation: Recall that the shard roster change algorithm gradually applies changes to the roster of a shard, one change at a time (see section 3.2). While this design ensures some security and liveness properties, it is slow and does not scale well since a transaction must be sent for each change. An improvement would be to keep gradually swapping-in new validators, however swapping-out old validators could be done in batches instead. The batch size is an important parameter since a large value implies a higher risk that the number of honest nodes remaining is too little to achieve consensus.

- Adding and removing nodes: Implementing new instructions to add or remove nodes from the identity byzcoin.

- Sign-up scheme: Currently, only the creator of an omniledger (admin or owner) can request a new epoch (in future work, the admin would also be able to add or remove nodes from the identity byzcoin). A possible improvement would be to develop a sign-up scheme so that nodes different from the admin can change the IB roster. An main idea would be to allow admins to delegate trust to a group of nodes. Members of this group can cast their vote to any node and will decide which node will participate in the next epoch. Potential improvement

on the scheme include integrating Proof-of-Personhood[12] as a membership mechanism to prevent Sybil attacks or implementing a form of Proof-of-Stake[8] in the voting process.

- Trust-but-Verify transaction validation: Omniledger proposes a two-level validation scheme in order to reduce validation time. The main idea is to use two groups of validators, optimistic validators and core validators. A transaction will first be sent to an optimisic validator shard for a first verification, then to a core validator shard for a second and final verification. The difference between the two lies in the shard size, an optimistic validator shard will be much smaller than a core validator shard. Smaller shards can process transactions faster but are less secure (the smaller the shard, the smaller the number of nodes in the shard an adversary needs to control). This scheme provides flexibility for clients as they can choose to wait for both validation steps for maximum security or only the first in case of low-value payments.

- Trust-but-Verify transaction validation: As mentioned in the introduction (see section 1) Omniledger proposes a two-level validation scheme in order to reduce validation time where a transaction is first sent to a an optimistic validator shard before being sent to a core validator shard. Optimistic validators provide faster but less secure validation since their shard is much smaller than the core shard. This scheme provides flexibility for clients as they can choose to wait for both validation steps for maximum security or only the first in case of low-value payments.

6 Installation

Note: This project's code is currently on a branch. It is planned to be merged into the main branch. The instructions below assume the code has been merged.

A working installation of Go is required to run OmniLedger. The project is in the `omniledger` directory of Cothority. The Cothority code, which contains the code of this project, is freely available to the public at `https://github.com/dedis/cothority`. The Go package and its dependencies can be downloaded and installed by using the following commands: `go get github.com/dedis/cothority`, `go get github.com/dedis/onet`, `go get github.com/dedis/kyber`.

References

[1] github.com/dedis/onet/tree/master/app#libtestsh.

[2] https://github.com/dedis/cothority/blob/master/byzcoin/Contracts.md.

[3] github.com/dedis/onet/tree/master/app#libtestsh.

[4] Cothority network library. https://github.com/dedis/onet.

[5] DEDIS – Decentralized and Distributed Systems. dedis.epfl.ch/.

[6] dedis/cothority: Scalable collective authority. github.com/dedis/cothority.

[7] A Distributed Randomness Beacon Daemon. github.com/dedis/drand.

[8] Proof-of-stake, wikipedia. en.wikipedia.org/wiki/Proof-of-stake.

[9] Reflection-based Protocol Buffers for Go. github.com/dedis/protobuf.

[10] stretchr/testify: A toolkit with common assertions and mocks that plays nicely with the standard library. github.com/stretchr/testify.

[11] testing - the go programming language. golang.org/pkg/testing/.

[12] Maria Borge, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, and Bryan Ford. Proof-of-Personhood: Redemocratizing Permissionless Cryptocurrencies. In *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 23–26, Paris, April 2017. IEEE.

[13] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. *arXiv:1602.06997 [cs]*, February 2016. arXiv: 1602.06997.

[14] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598, San Francisco, CA, May 2018. IEEE.

[15] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J. Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. Cryptology ePrint Archive, Report 2016/1067, 2016. https://eprint.iacr.org/2016/1067.

A Protocol diagrams

This section contains diagrams showing the communication flow of some protocols from section 3.

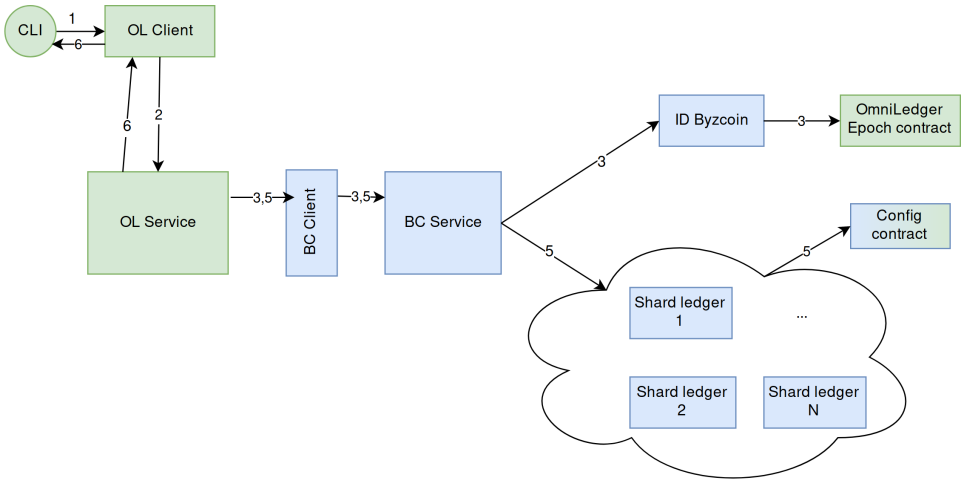

Figure 4: Communication flow between the different components during the execution of a new OmniLedger creation. The numbers indicate in which step the communication occured.

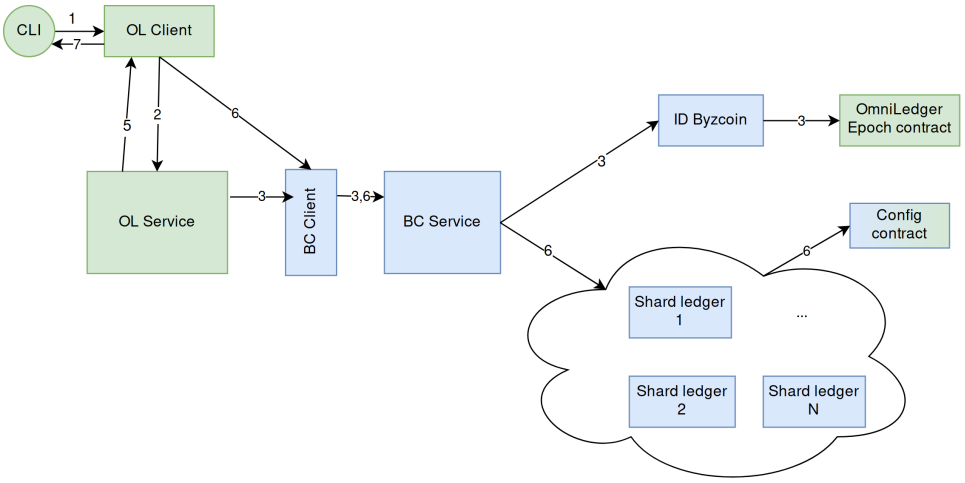

Figure 5: Communication flow between the different components during the execution of a new epoch request. The numbers indicate in which step the communication occured.

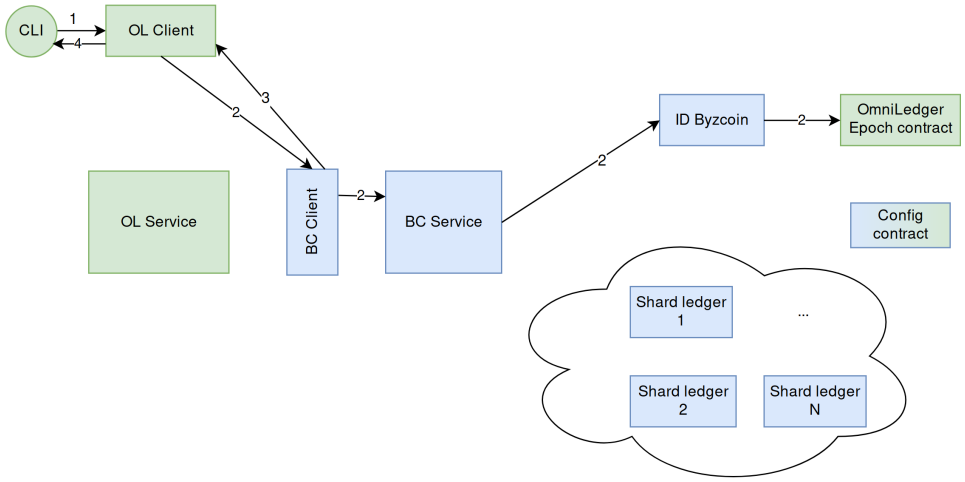

Figure 6: Communication flow between the different components during the execution of a get status request. The numbers indicate in which step the communication occured.