ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

D-VOTING – ENGINEERING A PRODUCTION-READY SYSTEM FOR AN E-VOTING PLATFORM

EPFL IC DEDIS
BC 210 (Bâtiment BC)
Station 14
CH-1015 Lausanne

by:
Igowa Giovanni (Master Semester Project)
Zhang Guanyu (Master Semester Project)

Project Advisor:
Prof. Dr. Bryan Ford (DEDIS)
Project Supervisor:
Noémien Kocher (DEDIS)

10th June 2022

Abstract

Electronic voting system has drawn more and more attention in the recent years. DEDIS lab has initiated a distributed voting system based on DELA[1] blockchain, called *D-voting*. After a couple of iterations, the *D-voting* system becomes a functional prototype[2] ready for deployment. In our work, we first went thoroughly through the existing system to find critical steps. We then put in place a testing pipeline on top which our tests are executed. Afterwards we implemented a series of automatic tests to fully test the correctness, performance and robustness of the d-voting system in realistic scenarios. At the end, we investigated the errors occurring during the tests and proposed some modifications and other ways for further investigations in order to bring the system fully production ready.

Contents

CHAPTER 1

INTRODUCTION

In this chapter, we will first review in detail critical components of the D-voting system and DELA Distributed Ledger service on top on which the D-voting system is build, analyse their potential limitations in a real production environment, and present the test framework and our testing methodology.

1.1 SYSTEM DESCRIPTION

The complete description of the system specification and election process can be found in our predecessors' report[2]. In our work, significant efforts have been involved in the first phase to understand the full functioning of the D-voting system and make hypothesis on what may go wrong in a real production environment and build our tests accordingly.

1.1.1 SECURITY SPECIFICATIONS

The D-voting needs to satisfy the following security standard in a real production environment. It is demonstrated by our predecessors that those aspects are theoretically respected.

- **Ballot secrecy** Vote content is encrypted. DKG can assure that.
- **Voter anonymity** Voter is not associated to their vote. Neff shuffle service ensures the voter anonymity.
- **Data integrity** Votes are securely stored and cannot be tampered with or deleted.

1.1.2 E-VOTING SMART CONTRACT

The election process is executed under the form of a smart contract running on a high level blockchain *DELA* developed by DEDIS[1]. Data are stored in each node of the blockchain and different election steps, like opening, casting a vote and closing are realised by adding transactions to the chain.

In order to decrease the network overhead in DELA, bidirectional stream-based protocol is implemented. An orchestrator is used to handle communications between nodes. But in a real production environment, network can be heavy or simply nodes can fail. In the worst case, the nodes can fail to elect a new leader and causing the malfunctioning of the D-voting layer. This case should be studied and tested in detail to understand whether the theoretical fault-tolerance is working.

1.1.3 DKG CRYPTOGRAPHIC SERVICE

The D-voting system implemented Distributed Key Generation (DKG)[3] mechanism to ensure the secrecy of votes. DKG generates a key pair in a distributed system: the public key is unmodified and available to all actors, but the private key is divided into shares and distributed to all actors of the system. One uses the public key for encrypting the content of the votes. After the shuffling step which dissociates the voters and theirs votes, a threshold of private shares are requested to decrypt the voting results.

In the last iteration, decryption phase takes exponential time with respect to vote numbers. And this could be the overall performance bottleneck of the system.

1.1.4 NEFF SHUFFLE SERVICE

The Neff shuffle algorithm[4] is implemented in D-voting system in order to dissociates the voter and their votes. The ballots sequence are shuffled by a threshold of nodes and it is possible for a verifier to verify that this shuffling process is indeed done.

In the shuffling phase, $\frac{2}{3} + 1$ rounds of shuffling need to be done which would solicit a large number of communication between nodes. When deployed on a real environment with multiple nodes, the D-voting system can stop from working.

1.2 TESTING PIPELINE AND METHODOLOGY

Once we targeted all critical phases and possible errors, we need to build a test frame work and methodology which allows us to investigate in a total, complete and general manner different aspects of the D-voting system.

In our work, in the second phase, we dedicated ourselves to build a testing pipeline consisting in: launching d-voting nodes on a user-defined deployment environment, setting up the system, running test, collecting and analysing log files. This framework is very important because it allows us to get consistent and reproducible results and log files. And consistency and reproducibility are extremely important when debugging a distributed system deployed on multiple nodes as randomness and abstraction make it hard to really understand what it's going on under the hood.

More precisely the following aspects of the D-voting system are tested in our work:

- Correctness: We simulate a whole election process from opening by an administrator to voting by a standard user from end to end when voting. All actions are triggered by proxy APIs which are the real endpoint of the system.
- Scalability: In the current EPFL e-voting system, one section hosts one node. So the D-voting system need to be deployed on at least 13 nodes, which correspond to the 13 sections in EPFL. And in order to guarantee the stability in a production environment, we need to test it on more nodes.
- Resilience: D-voting system needs to be working even though a couple of hosting servers are down. Theoretically, the D-voting system is still functional with only $\frac{2}{3} + 1$ nodes up, according to DELA specifications and the implementation of the DKG and the Neff shuffle services. We will simulate a situation where nodes are down to see whether voters can continue to vote, the shuffling can be done, decryption can be done. We will also test whether the maximum down node numbers correspond to its theoretical value.
- Integrity: If all nodes are down, we need to recover the election data when nodes are up again. And if a node is back during one specific step, the node needs to be able to update its state to the current newest state.

- Performance: In a real election setting, we can imagine that there are multiple elections going on simultaneously an D-voting system. Up to 20, 000 (roughly the total number of students and stuff at EPFL) votes can be cast in the biggest elections. It means the D-voting system needs to be able to handle a high level of access.

In the third and last phase of our work, we implemented tests corresponding to different aspects of the system in our testing pipeline, explained the results and log files in detail, proposed certain modifications, and showed ways for further investigations.

CHAPTER 2

METHODS

2.1 TEST DEPLOYMENT

Until last iteration, all tests are deployed on one local machine: nodes of the distributed system are launched as processes in one operating system. But it presents a couple of flaws majorly as it's very different from a realistic production environment where nodes are deployed in different servers and communications between those servers are done through real network where traffic can be heavy and debit low. And docker container can be a better choice.

2.1.1 DOCKER CONTAINER

We decided to deploy the D-voting system on docker containers to execute all our tests. This is motivated by the following advantages of docker environment:

- Network communication is slower than directly deployed on local machine. And it's possible to regulate individual node's bandwidth to simulate a real network lag.
- It demands less resources than deploying on traditional virtual machine environment.
- Docker container is an industrial standard deployment tool. Even though we test the D-voting system in our PC. The environment is very close to the production environment.
- The deployment is very fast and it is ease to create new instances. It makes it easier to get statistics of the success rate and errors in different settings.
- Docker containers allow us to simulate node down during the election process and bring it back again into the blockchain.
- The deployment on docker will not change existing test scripts as docker can map ports to localhost addresses which are used in all test scripts.

The docker file used for image build is very straightforward:

```
FROM golang:1.17.7-alpine AS build

ENV PATH="${GOPATH}/bin:${PATH}"
ENV LLVL=info

COPY . /d-voting
WORKDIR /d-voting/dela/cli/crypto
RUN go install

WORKDIR /d-voting/cli/memcoin/
RUN go install

EXPOSE 2001
WORKDIR /d-voting
```

We equally wrote two scripts `runNode.sh` and `setupnNode.sh` to launch and set up automatically the D-voting system with user defined number of nodes on the blockchain. Another script `autotest.sh` allows to automatise the launch, setup, test running, and log saving of the system so that we can get a larger number of logs which is very useful to debug a decentralised system where a lot of things can happen simultaneously under the hood.

2.1.2 COMPATIBILITY

The scenario test can be used when the node are running in local or in docker container mode. As each node expose a proxy address which is mapped to a localhost port in the same address space. In local mode, it's directly realised with memcoin utility. In the case of docker container, it's realised with the docker port mapping feature. All tests trigger different actions by sending HTTP requests to proxy APIs. As the address space is identical in the two cases, tests are by default compatible with both local test and docker test.

In this case the requests are sent to the local address of proxy. To have a better idea of the election process behaviour in real situation we have choose to isolate node on the docker. Each dela node that we will use to run election process will listen request from separated docker and we can use the scenario test to follow the good progress of the election. So the scenario test is compatible with both local test and docker test.

2.2 SCENARIO TEST

GOAL

The scenario test simulates a mocked election with some mocked votes for end to end. If the election passes the scenario test, it means we can run correctly an election in basic setting.

An important step in our project was to implemented the scenario test. Indeed it allow us to test the election process and show it is correct: here we mean the decrypted votes are identical to votes casted in the frontend. And the scenario test is also the base for introducing other kind of tests such as resilience test, performance test, and scalability test.

The scenario test simulates all different steps of elections process, which we present here in list:

- Create election
- Setup DKG

- Open election
- Get election info
- Cast ballots
- Close election
- Shuffle ballots
- Request public shares
- Decrypt ballots

Our implementation of scenario test differentiates from our predecessor's in six ways:

- All transactions are sent exclusively via node proxy APIs which are the standard endpoints during one election. Web frontend communicates with D-voting backend via those APIs. In the last iteration, some backend codes are involved. So the test is more realistic and isolated.
- It allows to run multiple elections simultaneously and cast any number of votes.
- HTTP requests are sent randomly to one of the proxy address of the distributed system. It is the expected behavior of web front end. The randomness is used to reduce the probability to send a request to a malicious node.
- HTTP requests now are signed before sent to proxies. It is the expected behaviour of frontend. Because the authentication is done on frontend with the help of `Tequila` service.
- Election results obtained at the end of the decryption process are compared exhaustively with the votes casted in the frontend in order to ensure not only decryption is successful but also the contents of the votes are not modified by a third party. A pooling mechanism is implemented in order to ensure election status is correct before triggering new actions.

As the scenario test is such a basic correctness test that all changes in D-voting system codes should not influence its result given that proxy APIs remain unchanged. We added it to GitHub continuous integration tests as a basic sanitary check.

REQUEST SIGNATURE

We put in place a critical security feature. Now the nodes proxies have been updated to accept only signed request. The signature can only be done by authenticated users by the frontend with the help of `Tequila` service. In comparison, in the last iteration everyone can send request to proxies as they are public. This feature allow also to connect the backend and the frontend, now they can communicate, which what not the case in the last iteration.

We use the Schnorr signature from the cryptographic package *kyber* [5] developed by DEDIS.

The function that we use in scenario test to create a signed request is as follow:

```
func createSignedRequest(secret kyber.Scalar, msg interface{})
([]byte, error) {

jsonMsg, err := json.Marshal(msg)
if err != nil {
return nil, xerrors.Errorf("failed to marshal json: %v", err)
}

payload := base64.URLEncoding.EncodeToString(jsonMsg)
hash := sha256.New()
hash.Write([]byte(payload))
md := hash.Sum(nil)
signature, err := schnorr.Sign(suite, secret, md)
if err != nil {
return nil, xerrors.Errorf("failed to sign: %v", err)
}

signed := ptypes.SignedRequest{
Payload:   payload,
Signature: hex.EncodeToString(signature),
}

signedJSON, err := json.Marshal(signed)
if err != nil {
return nil, xerrors.Errorf("failed to create json signed: %v", err)
}

return signedJSON, nil
}
```

First the message is marshaled to a Json message , then we encode it and with the secret key we can sign the hash of the encoding payload. We use the hash for the signature to avoid that two similar Json message have the different signature. When the proxy receive a new request, it can check the signature with the public key. Our scenario test, which plays the role of the web-backend(charged for user authentication), knows the secret key and can send sign request to the node proxy.

RANDOM PROXY REQUEST

In the production environment, the frontend can send HTTP requests to any node proxies stored in the proxy array to trigger actions. Similarly, in the scenario test we store a list of proxy node address and every request is sent to one address in the list at random. Therefore, we can prove that any proxy node works and can be used. In case of the presence of malicious nodes, the probability of sending a request to a malicious node is small thanks to the randomness.

```
randomproxy = proxyArray[rand.Intn(len(proxyArray))]
```

ELECTION STATUS CHECK

Actions triggered by different transactions on the block are supposed to bring the election to different status. The different status are as follow:

```
Initial Status = 0
// Open is when the election is open, i.e. it fetched the public
    key
Open Status = 1
// Closed is when no more users can cast ballots
Closed Status = 2
// ShuffledBallots is when the ballots have been shuffled
ShuffledBallots Status = 3
// PubSharesSubmitted is when we have enough shares to decrypt
    the ballots
PubSharesSubmitted Status = 4
// ResultAvailable is when the ballots have been decrypted
ResultAvailable Status = 5
// Canceled is when the election has been cancel
Canceled Status = 6
```

One issue in a distributed system like D-voting is that status can be not up to date in all nodes. But it's crucial to go into the right status before triggering next actions. Otherwise, errors can happen or action will be ignored. What's more, status allow us to see whether actions succeed.If we get the right status after an action we can assume that the output of the action indeed takes place. For example shuffle or open election have worked correctly before going to the next step.

In the last iteration, in order to guarantee all nodes reach the correct status, `time.Sleep` is used before each new action in the scenario test. With the purpose of avoiding using an excessive number of `time.Sleep` in the scenario test, which would slow down unnecessarily the test and introduce bias when measuring the performance of the system, we implemented a polling mechanism: the test is blocked before the election status becomes correct.

VOTING RESULT TEST

In the previous iteration, no proper test is implemented to guarantee the content of the votes after decryption is strictly identical to the content of the votes casted by the voters on the user frontend. We implemented a simple mechanism to verify the correctness of the encryption/decryption process with a complexity of $\mathcal{O}(n^2)$. Each decrypted vote is compared to the list of casted votes. This process is feasible in a real election scenario in which the total number of votes is upper bounded by around 10,000 which corresponds to the total number of personal and students in EPFL. Therefore, we can test the correctness of a election process by first cast ballot, let them pass through all other process and finally check after the description process if the list of ballots cast are the same as the list of decrypted ballots.

EXPECTED RESULT

If the whole election process goes well, it's expected that the test is passed with no error message. Otherwise, we handle all errors in the process and the error messages will allow us to understand the system under the hood and debug the system.

2.3 SCALABILITY TEST

GOAL

Scalability is an important property when developing a distributed application or system. It is the ability or the capacity of the system to increase in performance and cost in response to changes of user demand. In our case, in the previous iterations, the vast majority of tests were concentrated on a small system setup: the testing system is deployed on 3 nodes and we casted 3 votes in one single election. But in the real production environment, much more votes would be cast during one election and there might be multiple elections simultaneously. Until now the maximum number of ballot used in a test is 50. We want to see if the system can scale with much more ballots which represents the case of a big election with a high number of participants. What's more, the system is likely to be deployed on more nodes in order to guarantee security. For example, a typical use case for the EPFL will be to use 13 nodes, one per section. Thus the goal of scalability test is to show that the system is able to scale well with respect to this 3 aspects: node number, ballot number, and election number.

IMPLEMENTATION

In order to run tests a large number of times automatically and get statistics of the system state, we wrote a bash script `autotest.sh`. And the tests are deployed both on local machine and on docker container.

Firstly, we scale with respect to the node number. The scenario test mentioned above will be used for this purpose. Indeed it allows us to see if all election steps correctly end and if the ballots cast in the front end are the same as the decrypted ones. Therefore the idea is to run 15 times the scenario test for each different number of nodes and get the average success rate for each case. We will test 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 20, 25, 50 nodes. The idea is to increment the number of nodes progressively until reaching an error. In case where no error occurs, the upper limit will be set to 200 nodes because in our use case i.d. internal election at EPFL, it's highly impossible the system will be deployed on more than 200 nodes. We are expecting that errors will occur when node number exceed a certain threshold due to communication overhead between nodes.

Secondly, we scale with respect to the ballot number. For the ballot number we will try: 3, 10, 50, 100, 200, 500, 1000, 2000. Again the idea is to increment number progressively and see if we encounter an error. The expectation for this part is that we can have a communication problem when the number of ballots grows because nodes need to send a huge number of messages to each other to cast the ballot.

2.4 RESILIENCE TEST

GOAL

Another important property that we need to test is the resilience of the system. Resilience can be defined in our case as the capability to correctly end up the election process even if one or more node fail during any steps of the process. And in case when nodes are back to the network, their states can be correctly updated to the newest.

IMPLEMENTATION

In the D-voting project, the simplest way to simulate node failure is to kill some nodes during the election process as nodes are either run as process if deployed in local machine or containers if deployed in docker. For both cases, we implemented a function to allow testers to kill specific nodes during the process. What's more, to be closer to a real case where any of the nodes can fail, nodes can chose at random to be

killed. We want to see whether an election process could end correctly when a certain number of nodes is down.

Again the aforementioned scenario test will be used for this purpose. We can distinct local and docker deployment cases. With the scenario test, the correctness of execution can be monitored after each step. For each node number, we run the test 15 time and compute the success rate. First, with the number of nodes 4, 5, 6, 7, 8, 9, 10, 11, 12, and 13 we will kill one node and then with the number of nodes 7, 8, 9, 10, 11, 12, 13, 14, we will kill two nodes. Afterwards we'll test whether the theoretical threshold of system correctness corresponds to reality by killing $\frac{1}{3} - 1$ nodes. We will kill nodes before each critical steps of the election process: cast ballots, shuffle ballots, and request public shares. It is expected that if the number of living nodes exceeds the theoretical threshold $\frac{2}{3} + 1$, the election process will finish with the correct result.

In addition, we also want to simulate the case where a failed node rejoin the network after a certain time and test whether the node can be rejoin the network without its data being corrupted and the election process could end correctly. If this property holds, when more than $\frac{1}{3} - 1$ would go down, the election process could resume after nodes rejoin the network. Bringing back a node can be easily realised on docker with the command `docker restart`.Now the question would be whether is possible to revive nodes at any state and correctly finish the process? To have a better idea of the system execution we test different cases. We will kill nodes before each critical steps of the election process: cast ballots, shuffle ballots, and request public shares and resuscitate them after a random amount of time. It's expected that election will succeed as normal when the number of living nodes is above $\frac{2}{3} + 1$ and election will take more time to finish otherwise. The extra is correlated to the waiting time before bringing back the node.

2.5 PERFORMANCE TEST

GOAL

It's expected that the execution time will increase when the system scales up. But D-voting system must not only produce the right output but also do it in a reasonable time. Because in a production environment, a long waiting time for one vote to be accepted may decrease the participation rate in an election. So a high response time will restraint system's utility.

The last D-voting backend team has already given us an idea of this execution time in their report [2] with `performance_test.go`. But one major flaw is that actions are not triggered by HTTP requests as in the production environment. In our work, we have done the measure differently, we use a scenario test which acts as a proxy for user web frontend. The scenario test uses HTTP requests to communicate with the node proxy. In this way, we are closer to the real use case. The measurement is done by setting a timer at the beginning and the end of each step of interest in the scenario test.

SHUFFLING STEP

The main critical operation that can take a long time is shuffling. The shuffling time depends on the number of nodes and ballots as highlighted in [2], so we will measure the shuffling execution time with an increasing number of nodes and ballots. The tests are done with 3, 5, 7 nodes and 3, 10, 25, and 50 ballots both in local and docker environment. It are expected that the shuffling time grows linearly with the number of nodes and ballots and that it takes more time on docker than in local because the communication is much faster in local.

DECRYPTION STEP

In the last iteration, the decryption time increases exponentially with the number of ballots due to certain implementation choices. The high computation cost will reduce the usability in a giant election. In mid semester, the computation logic is modified so that theoretically the decryption time grows linearly with respect to ballot number. We'll measure and demonstrate it.

TOTAL ELECTION TIME

Other steps in the election process is linear in time with respect to node number except the DKG setup which is exponential to node number but this step is located before users cast their votes so it has less impact on the usability of system. So the total election time is limited by shuffling and decryption steps.

CHAPTER 3

RESULTS AND EVALUATION

In order to guarantee test consistency, all tests are executed on a machine equipped with *intel core i7* CPU and running *Ubuntu 20.04.5 LTS* operating system.

3.1 ELECTION PROCESS CORRECTNESS AND SCALABILITY

3.1.1 GOAL

The goal of this part is to test on local and on docker environments a simple election process. We want to use our scenario test to show that an election can run and output the right result.

We also want to find the limitation of the system in terms of number of nodes and number of ballots. Two questions we would like to ask are: on how many nodes can the D-voting system be deployed and what is the maximum number of ballots that we can cast.

In this section, we have run a series of tests with different numbers of nodes: 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 20, and 25. Firstly, we deployed the system locally, and then we deploy the system on docker containers. We will also run an election with the different numbers of ballots with a fixed number of nodes: 3, 10, 25, 50, 100, 150, and 200. We will run 15 times each test for test case. The time is the average time of all tests.

3.1.2 NODE LIMITATION

LOCAL

In a local deployment, while varying from 3 to 15 nodes we encounter no problem, the election process runs normally and gives us the right result 3.1. Apart from 20 nodes the result is different. We reach the scenario test timeout which is fixed to 600s. By analysing all log files, we deduced the following error pattern which occurred in a majority of nodes:

Number of nodes n	**Success rate test** 15 attempts	**Time (in seconds)**
3	100%	59.93s
4	100%	66.02s
5	100%	73.37s
6	100%	92.81
7	100%	98.78
8	100%	115.97s
9	100%	135.77s
10	100%	127.89s
12	100%	169.73s
15	100 %	198.19s
20	FAIL timeout test == 10m %	600s
25	FAIL timeout test == 10m %	600s

TABLE 3.1
The evolution of scenario test success rate with the number of nodes (local)

```
/core/ordering/cosipbft/mod.go:515 >
view propagation failure error="failed to call
client: rpc error: code = DeadlineExceeded desc =
context deadline exceeded" addr=localhost:2016

/core/ordering/cosipbft/proc.go:165 >
view message refused
error="invalid view:mismatch leader 7 != 12" addr=localhost:2016
```

DOCKER

Number of nodes n	**Success rate test** 15 attempts	**Time (in seconds)**
3	100%	62.80s
4	100%	66.70s
5	100%	76.18s
6	100%	77.19s
7	100%	88.89s
8	100%	86.52s
9	100%	113.18s
10	100%	116.28s
12	100%	123.48s
15	100 %	147.98s
20	FAIL timeout test == 10m	600s
25	FAIL timeout test == 10m	600s

TABLE 3.2
The evolution of scenario test success rate with the number of nodes (on docker)

While deployed with docker, varying for 3 to 15 nodes we encounter no problem, the election process run

normally and give us the right result 3.2. Similarly to the case of local deployment, we find exactly the same problem apart from 20 nodes. We reach again the scenario test time out error. In the log files of nodes, we get the same recurrent error `invalid view:mismatch leader`.

ANALYSE

We can see that the number of nodes on which we can deploy the D-voting system and run an election correctly is bounded: when deployed on more than 20 nodes, error occurs. By analysing carefully log files generated by each DELA node, we realised that the error comes from the communication of the DELA nodes and occurs as soon as the election begins. Each node displayes the following error message: `invalid view:mismatch leader`, saying that there is a leader change and the node does not have the right leader information stored in local memory. During the election process, the leader node changes again and again which is not normal behavior and can be explained by the too slow communication between the nodes. Indeed, with a high number of nodes, the communication between them becomes slower and impulse sent by leader is not received by other nodes regularly. Hence they think that the actual leader doesn't do his work correctly and they try to elect a new leader.

The repetition of this leader election prevents transactions from correctly being executed. In the first steps of the election process, the communication overhead is relatively light. So from time to time, the election process could pass those steps successfully. But the shuffle step never ends and leads to a "fail timeout test". During the shuffling phase, the node could successful send start shuffling message, creating transactions, and adding them to the pool. The problem comes from the fact that the transactions in the pool are not added to the blockchain, most of the time they are all denied. Therefore we cannot reach the mandatory shuffle threshold and we end up with a timeout. Unfortunately, we will see that the same leader election problem occurs in other test results.

With the scenario test, we have shown that an election can be correctly processed with a different number of nodes, we have also shown that this number is limited to 20 due to node communication. Our expectation was right on the node limitation but we don't expect that the limitation was small like that.

3.1.3 BALLOTS LIMITATION

LOCAL

After running elections on a D-voting system deployed locally with 3, 5, and nodes, varying number of ballots, we get as result this table 3.3.

# ballots n	**Success rate test /Time (in seconds)** 3 nodes	**Success rate test /Time (in seconds)** 5 nodes	**Success rate test /Time (in seconds)** 10 nodes
3	100% / 47.63s	100% / 59.93s	100% / 90.93s
10	100% / 67.91s	100% / 78.87	100% / 100.22s
25	100% / 86.95s	100% / 112.91s	100% / 183.29s
50	100% / 104.06s	100% / 154.64s	100% / 213.768854116
100	100% / 223.95s	100% / 303.51s	FAIL timeout test == 10m / 600s
150	100% / 360.40s	FAIL timeout test == 10m / 600s	FAIL timeout test == 10m / 600s
200	FAIL timeout test == 10m / 600s	FAIL timeout test == 10m / 600s	FAIL timeout test == 10m / 600s

TABLE 3.3
The evolution of scenario test success rate with the number of ballots (local)

DOCKER

After running elections on a D-voting system deployed with docker containers with 3, 5, and nodes, varying number of ballots, we get as result this table 3.4.

# ballots n	Success rate test /Time (in seconds) 3 nodes	Success rate test /Time (in seconds) 5 nodes	Success rate test /Time (in seconds) 10 nodes
3	100%/ 71.72s	100% / 74.26s	100% / 154.91s
10	100% / 65.48s	100% / 95.88s	100% / 191.17s
25	100% / 107.63s	100% / 160.25s	100%/ 197.15s
50	100% / 178.39s	100% / 294.92s	FAIL timeout test == 10m /600s
100	100% / 331.22s	FAIL timeout test == 10m /600s	FAIL timeout test == 10m /600s
150	100% / 367.62s	FAIL timeout test == 10m /600s	FAIL timeout test == 10m /600s
200	FAIL timeout test == 10m / 600s	FAIL timeout test == 10m / 600s	FAIL timeout test == 10m / 600s

TABLE 3.4
The evolution of scenario test success rate with the number of ballots (on docker)

ANALYSE

We can see that the number of ballots that we can cast varies with the number of nodes. If we deploy with fewer nodes, for example 3, we can cast a large number of ballots 150 compare to, for example 10 nodes, where the max cast ballots are 25. The error encountered when casting too munch ballots is a test timeout error. This error could be explained by the fact that, shuffling time is a linear function in node number and ballot number. So if we have a lot of nodes and a big number of ballots, the necessary time to finish the shuffling step will exceed 10min. But if we change this time to have a greater value we could cast more ballots without error. Therefore in a real situation where we don't limit the time to finish each step, we can bypass this error and cast a large number of ballots.

3.2 RESILIENCE TEST

3.2.1 GOAL

In this part, we want to test the resilience of the system. The D-voting system should be able to correctly run an election process even if some nodes are down during the process. According to DELA and D-voting implementation specification, if the number of living nodes is above $\frac{2}{3} + 1$ the election can be done correctly. As explained in the method section, firstly, with the number of nodes 4, 5, 6, 7, 8, 9, 10, 11, 12, and 13 we will kill one node and then with the number of nodes 7, 8, 9, 10, 11, 12, 13, 14, we will kill two nodes. In order to get consistent result, we execute each test 15 times.

3.2.2 THRESHOLD

The failure tolerant threshold used by the shuffle and the request public share steps is upper bounded by DELA node specification. A helper function is implemented to yield this number directly. For example, if we have 4, 5, and 6 nodes the output will be respectively 3, 4, and 5 which means the system is supposed to continue working while one node is down. If we have 7, 8 and 9 nodes the output will be respectively 5, 6, and 7 nodes which means the system can tolerate 2 node failures.

```
func ByzantineThreshold(n int) int {
if n <= 0 {
return 0
}
f := (n - 1) / 3
return n - f
}
```

3.2.3 LOCAL TEST

Number of nodes n	Shuffle success rate 15 attempts	Test success rate 15 attempts
4	100%	100%
5	100%	100%
6	100%	100%
7	100%	100%
8	100%	100%
9	100%	100%
10	100%	100%
11	100%	100%
12	100%	100%
13	100%	100%

TABLE 3.5
Scenario 1: Shuffle step success and whole test success rate when killing one node (local)

Number of nodes n	Success rate shuffle 15 attempts	Success rate test 15 attempts
7	100%	100%
8	100%	100%
9	100%	100%
10	100%	100%
11	100%	100%
12	100%	100%
13	100%	100%
14	100%	100%
15	100%	100%
16	100%	100%

TABLE 3.6
Scenario 2: Shuffle step success and whole test success rate when killing two nodes (local)

SCENARIO 1

In this part, we run the scenario test with different numbers of nodes and we kill one node before the beginning of the shuffle step. We have computed the average success rate of shuffle step and the whole test and we get this table 3.5.

In all node log files except the one killed, we get a `connection refused` error when other nodes try to send messaged to killed node. This error doesn't interrupt the election process. Because the number of living nodes is above the byzantine threshold.

SCENARIO 2

In this part, we have run our scenario test with a different number of nodes and we have killed two nodes before the beginning of the shuffle step. We have computed the average of the test output and we get this table 3.6.

As in the first scenario, the `connection refused` error occurred again. But the election process can correctly finish as expected.

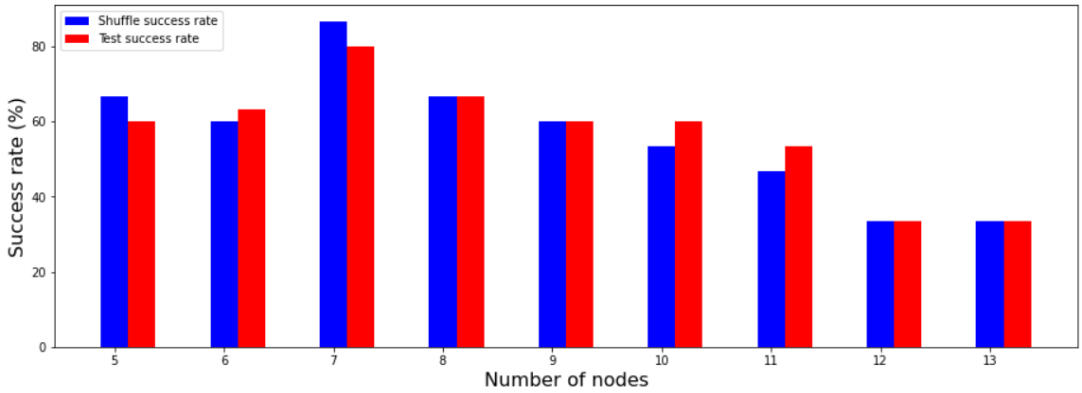

FIGURE 3.1
The evolution of the shuffle and scenario test success rate with the number of nodes in the case a node is kill before the shuffle step.

3.2.4 DOCKER TEST

Number of node n	**Success rate shuffle** 15 attempts	**Success rate test** 15 attempts
3	0%	0%
5	66.7% (10/15)	60% (9/15)
6	60% (9/15)	63.3% (8/15)
7	86,7% (13/15)	80 % (12/15)
8	66.7% (10/15)	66.7% (10/15)%
9	60% (9/15)	60% (9/15)
10	53.3% (8/15)	60% (9/15)
11	46.7% (7/15)	53.3% (8/15)
12	33.3% (5/15)	33.3% (5/15)
13	33.3%(5/15)	33.3% (5/15)

TABLE 3.7
Scenario 1: Shuffle step success and whole test success rate when killing one node (docker)

SCENARIO 1

In the first scenario, we will kill only one node during the election process just before the beginning of shuffle step. All the nodes are running in separating containers which exposed the ports to localhost. The fact that IP address space is the same in local and docker deployment means we don't need to modify the test script in those two cases. To kill a node we can simply stop the container on which the node is running. It's expected that we get exactly the same results as in local deployment because nothing changes except the deployment environment.

After multiple runs, we get this table 3.7. In comparison to local deployment setting, we got heterogeneous results. The nodes seem to have more difficulty to collaborate. We find two different types of errors.

The first type of error is `threshold of shuffling not reached:`. For example in the case of 7 nodes, the error message is `threshold of shuffling not reached: 2 < 5`. The nodes cannot reach the shuffle threshold within a predefined number of tries. To go further, we modify this number from `number of nodes * 10` to `number of nodes * 20` to see if given more tries, the nodes can achieve sufficient rounds of shuffle. This change helps to improve a little bit the test success

rate but in most of the cases the nodes end up with the same error message. Despite additional attempts the nodes can still not reach the shuffle threshold. We'll further discuss this part in the last section of the report 4.2.

The second error that we find is during the request public shares step. Similar to the shuffle error, the node in charge of receiving public shares cannot reach threshold necessary to restore private key and decrypt the ballots and we end up with a test timeout error.

SCENARIO 2

Number of node n	**Shuffle step success rate** 5 attempts
7	(0/5)
8	(0/5)
9	(0/5)
10	(0/5)
11	(0/5)
12	(0/5)
13	(0/5)
14	(0/5)

TABLE 3.8
Scenario 2: Shuffle step success when killing two nodes (docker)

In the second scenario, we will kill two nodes during the election process just before the beginning of shuffle step. We expect the same behavior as in the local test: election ends successfully as failed nodes are under threshold.

The results of the tests are gathered in this table 3.8. Compared to the first scenario, the system can no longer correctly end the shuffle step no matter the node numbers. In this case, we have only one type of error which is the test timeout error. The shuffling process seems to be locked and doesn't make any progress. The system cannot even achieve one round of shuffling.

SCENARIO 3

Number of node n	**Election success rate** 5 attempts
7	(5/5)
8	(5/5)
9	(5/5)
10	(5/5)
11	(5/5)
12	(5/5)
13	(5/5)
14	(5/5)

TABLE 3.9
Scenario 3: Whole test success rate when killing two nodes and restart them (docker)

In the third scenario, we will kill two nodes during the election process just before the beginning of shuffle step and restart them 15 seconds afterwards. We expect that the shuffle step being blocked until nodes restart and reenter the network, and the election process can end successfully.

The results of the tests are gathered in this table 3.9. Compared to the second scenario, the system stopped being blocked in the shuffle step once failed nodes back to the network. The election process ended successfully but it took more time. The extra time is equal to the time between killing and restarting nodes.

ANALYSE

When deployed on docker containers, in both scenarios election process encounters a timeout error. By looking at nodes log files, we can have a clearer idea on what it's going in the system and find the origin of this problem. As soon as we killed a node or two, we meet first an `connection refused` error because the killed node is unreachable by all other node. Afterwards, we come through a `view change` error. This error is triggered when nodes fail to elect a new leader. We speculate that the fact of retrying to send messages to the unreachable node again and again slows than the communication of the whole network between all nodes. As consequence, the leader is considered as down for certain nodes and they start to elect a new leader. In certain occasions, nodes try to reelect the leader again and again which slows down the entire election process.

In scenario 1 the node which receives the HTTP request can send a shuffle message, create a shuffle transaction and add it to the pool. Then some of these transactions are added to the blockchain. Unfortunately, this process takes more time than before due to the leader error and therefore we end up with a `threshold of shuffling not reached:` error or a test timeout error. This is the same situation for the request public shares step, the nodes are not able to make some progress, and most of the transactions proposition in the pool are denied which leads to a test timeout error. In the graph 3.1 we can see that passing from 5 to 7 nodes the success rate grows a bit and from 8 to 13 nodes the success rate decreases again. We can speculate that the global trend of success rate is decreasing when we increase the node number. Because more nodes means slower communication between them. But when the node number is small, by increasing node number, it's more possible to reach the threshold of shuffling round.

In scenario 2, the situation is similar: leader election leads to the timeout error and 0 % success in the shuffling process. The zero success rate can be explained by the fact that the leader election error has more influence and the nodes spend more time trying to change the leader, instead of correctly executing the shuffle process in situations where we kill two nodes.

In scenario 3, we showed that the D-voting system is resilient to situation where failed nodes rejoin the network. One leader can be elected, the shuffle step can terminate, and election process continues.

By comparing deployment on docker and locally, we can say that the election process seems to better work on local deployment. It might be explained by the fact that the communication in local is faster than in docker deployment. And faster communication contributes to avoid the "leader change" problem. What's more, we trust blindly the docker implementation and it's possible that the error is caused by a bug in the docker implementation. Even though this scenario seems not very plausible.

Generally, the docker test reflects better the node behavior in real-world condition as the network bandwidth is constraint. Docker test also allows us to highlight the communication issue which would not necessarily be the case with only the local tests. By digging into errors present in log files, we can get inspired on how to further increase the quality of the D-voting system.

3.3 PERFORMANCE TEST

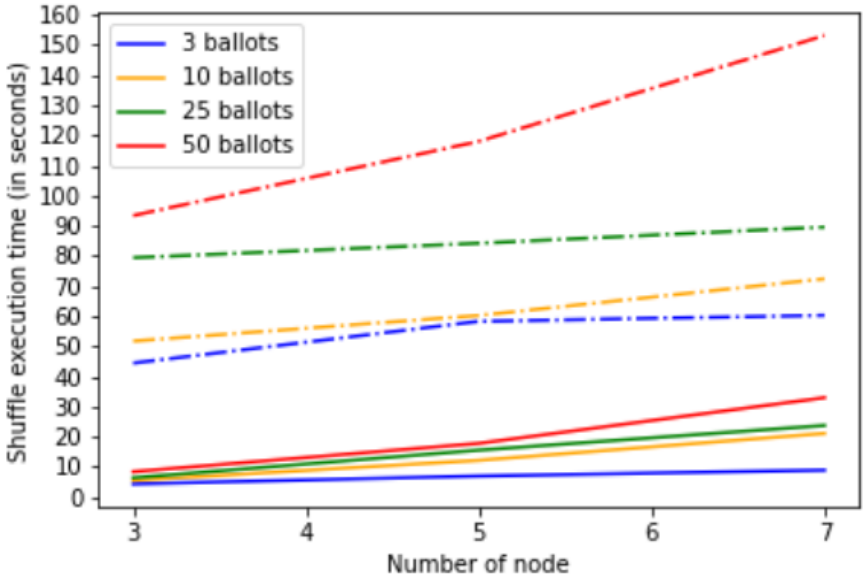

FIGURE 3.2
Evolution of shuffle execution time with # nodes and # ballots. The dotted lines represent the time measured on docker and the solid lines represent the time measured in local.

3.3.1 GOAL

The goal of this part is to evaluate the election execution time to see if we can run an election within a reasonable time. The election is composed of several steps and we will focus on the shuffle and decryption as they are most likely to be time-consuming. These several steps' execution speed depend on parameters like the number of nodes or ballots. Therefore we will measure different scenarios with different parameters as mentioned in the method section 2.5.

3.3.2 SHUFFLE EXECUTION TIME

LOCAL & DOCKER

In this part, we have measured the shuffle execution time with the different number of nodes : 3, 5, and 7. We have also varied the number of ballots with both local and docker deployment. The result is plotted here: figure 3.2. The first thing that we can see is that the shuffle execution with docker (in dotted lines) takes more time than in local and that the shuffle time is linear with respect to the number of nodes. This confirm our expectation 2.5: the more nodes we have, the more time the shuffle step will take. The behavior corresponds to our expectation because there are more transactions to be put in the blockchain blocks during the shuffle step which will take more time. We can also see that another factor on the execution time is the number of ballots. The more ballots we cast, the more time the shuffle step will take because the shuffling step has more ballots to shuffle.

ANALYSE

The result of this part allows us to say that we can run an election with relatively good time because the shuffle time is increased linearly with respect to node number and ballot numbers. Therefore the system parameter can grow and we can keep a good shuffle time. If we extrapolate the curve of 7 nodes in docker deployment 3.2, shuffling 10, 000 ballots would take roughly 8.3 hours which seems reasonable. From these results one might conclude that the shuffle time in real-world case would be good with big

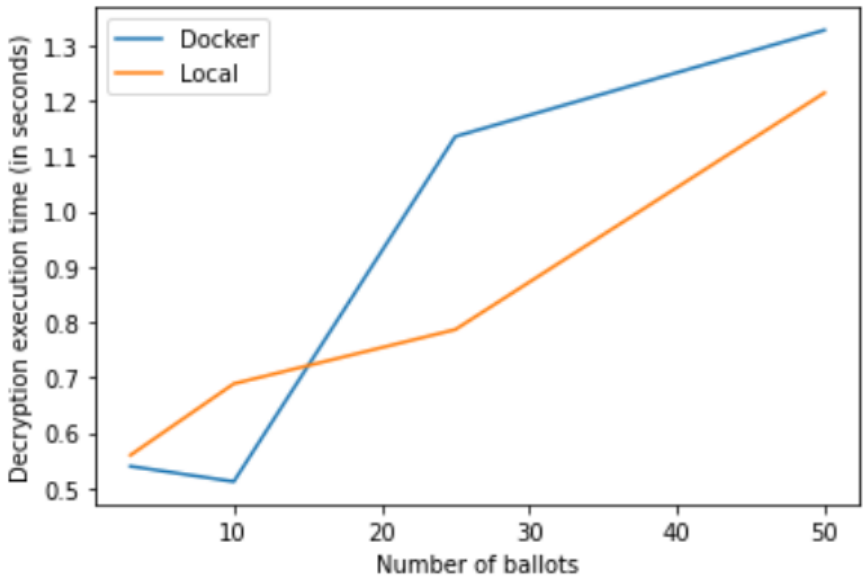

FIGURE 3.3
Decryption execution time w.r.t. number of ballots

parameters, however, a more detailed study on those two parameters in the section 3.1 and3.1.3 show us that from a certain parameter threshold the shuffle time will make the test timeout. Therefore, we can conclude that the system in terms of execution time is scalable up to a certain parameters threshold, for example, 50 ballots and 5 nodes 3.4

Note also that when several elections are run simultaneously in the local deployment, the execution time of shuffling take more time3.4 and grows linearly with the number of elections. But when deployed in docker, the shuffle time is identical when varying the number of elections. We'll further discuss this phenomena in section 4.1.

3.3.3 DECRYPTION EXECUTION TIME

ANALYSE

We have measured the decryption time as describe in section 2.5 and the result is plotted here: figure 3.3. Compared to the result of the last semester [2] where the decryption time increases exponential with respect to the number of ballots, we get a linear decryption time which varies between 0.5s and 1.3s. This is due to an update of the decryption process during the semester now the decryption is supposed to be linear with respect to ballot number. Therefore, the system ensure a reasonable decryption time even if the number of ballots grow which is important for the production ready system.

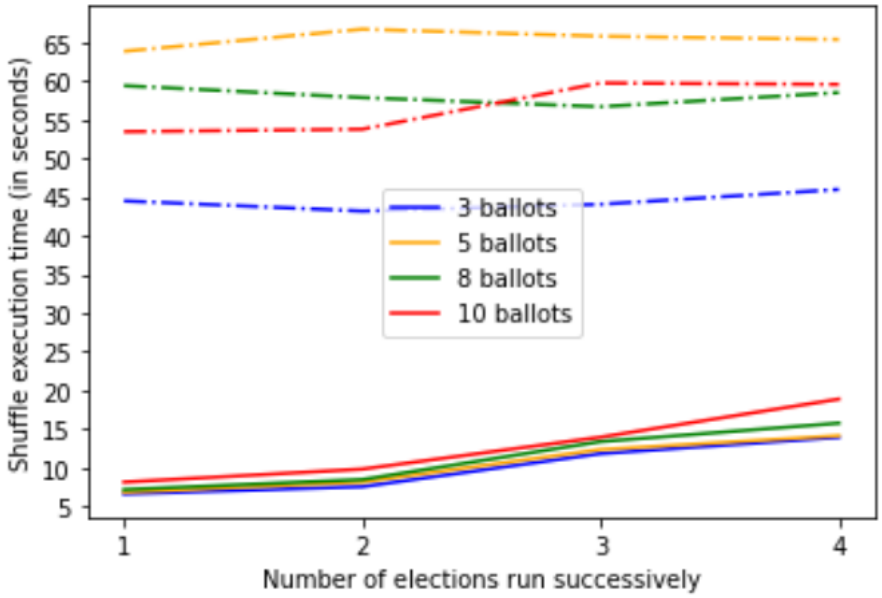

FIGURE 3.4
Shuffle execution time in function of number of elections run successively (3 nodes used)

CHAPTER 4

FUTURE WORK

4.1 DEPLOYMENT PLATFORM

During all our tests, we leverage the efficiency and ease of use of docker containers. This deployment choice is justified by its closeness to the real production environment: deployed on EPFL section servers. Indeed it allows us to explore errors hidden when the system is deployed locally. But in order to make sure that the system is working in production, there is no better way than really deploying it in a real production environment, i.d. on servers.

Otherwise, docker implementation is prone to errors as well. The strange curves of shuffle time with respect to election number in docker deployment in figure 3.4 are difficult to explain with arguments linked to D-voting system itself.

4.2 COMMUNICATION BETWEEN NODES

4.2.1 WAITING TIMES

NEFF SHUFFLE

The waiting time between checking shuffle transaction accept and retrying shuffle is hard coded. Now we coded it as a function of node numbers `4 + election.ShuffleThreshold/2`. The best waiting time needs to be found by a grid search.

REQUEST PUBLIC SHARES

Similar to the shuffle step, during the request public shares step, the waiting time between checking transaction accept and retrying is hard coded. Now we coded it as a function of node numbers `4 + election.ShuffleThreshold/2`. The best waiting time needs to be found by a grid search.

4.2.2 TREE TOPOLOGY COMMUNICATION

In DELA, we use MINO (Minimalistic Network Overlay) as an abstraction of a network overlay in which nodes are organised in a tree structure. This kind of algorithm is efficient in terms of distributed load but is very sensible to failures. In case where the failed node is the root or near the root, the election process will be blocked until the node is back to the network.

4.2.3 LEADER ELECTION

As we saw in section 3.2.4, when the communication is heavy i.d. high nodes numbers, high ballots numbers, certain nodes will lose connection to the current leader node and start electing new leader. This behavior might totally block the whole election process. One possible fix is to modify the waiting time between losing leader and electing a new leader as a function of node numbers. The actual relationship between waiting time and node numbers is hard to be determined theoretically. A grid search is necessary to find the best hyperparameters.

4.3 COMPARISON WITH OTHER E-VOTING PLATFORM

It's also interesting to compare the performance test result we got with results in other e-voting platforms like helios[6].

CHAPTER 5

CONCLUSION

In this work, we implemented a testing pipeline allowing us to get reproducible results, fully tested the correctness, scalability, resilience, and performance aspects of the *D-voting* system developed by our predecessors in a close to production environment, analysed the results, error messages and log files, proposed and tried certain modifications.

The capacity of D-voting system of running in a production environment is partially validated. More precisely:

- With the implementation of request signature verification, now the D-voting backend is supposed to be able to work in harmony with the web frontend.
- The D-voting system is functional when deployed on 13 nodes which corresponds to the use case in EPFL where we deploy one node on each section's server.
- The system can handle elections with a huge number of ballots in a reasonable time.
- When failed nodes go back to network, the election process can continue without errors.
- Theoretically D-voting system is able to handle up to $\frac{1}{3} - 1$ nodes failures. But in a close to real production environment, errors linked to communication overhead often occur.

However, the performance and resilience tests showed that the fault-tolerance of the system in the real production environment doesn't correspond to its theoretical performance. This aspect need to be rectified before putting D-voting system in a real production environment.

We deduced from log files that one major cause for the fault-tolerance issue is the communication overhead between nodes. We proposed three solutions to overcome this type of errors: change waiting time in Neff shuffle serving, change waiting time in DKG service (Request public shares step), and change the leader election mechanism. Actual functional hyper-parameters for those modifications need to be found in a grid search manner as it's hard to deduce a value theoretically.

RETROSPECTIVE

A significant effort has been put in the first phase of the project (three weeks) to understand the actual functioning of the D-voting system and DELA. Afterwards in the second phase of the project (five weeks), we implemented our deployment scripts, test pipeline and test scripts in several iterations. Lot of changes have been made between iterations because of different system updates. At the last phase, we iterated the system modifying, testing and results analysing circle numerous times. Testing the system for a large number of times is time consuming. And analysing more than 100 log files generated after each test is not

ease. Certainly, our proposed solutions with different wait time did not work at the end. But we think it's the right way to pursue to fix the errors.

PERSONAL CONCLUSION

Through this project, we fully understood the complexity of a modern distributed system and the difficulty of debugging it. Even though we did not fully validate the current D-voting system for a production environment and we did not solve the fault-tolerant under-performance issue correctly, we are positive that our test framework would allow our successors to easily test their solutions for errors that we have detected.

BIBLIOGRAPHY

[1] DEDIS. *Dela: Dedis Ledger Architecture*. URL: https://dedis.github.io/dela/ (visited on 2nd Mar. 2022).

[2] Emilien Duc Auguste Baum. *D-Voting: e-Voting on Dela*. URL: https://www.epfl.ch/labs/dedis/wp-content/uploads/2022/02/report-2021-3-baum-auguste-Dvoting.pdf (visited on 1st Mar. 2022).

[3] Torben Pryds Pedersen. 'A Threshold Cryptosystem without a Trusted Party'. In: *Proceedings of the 10th Annual International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT'91. Brighton, UK: Springer-Verlag, 1991, pp. 522–526. ISBN: 3540546200.

[4] C. Andrew Neff. *Verifiable mixing (shuffling) of ElGamal pairs*. URL: http://courses.csail.mit.edu/6.897/spring04/Neff-2004-04-21-ElGamalShuffles.pdf (visited on 2nd Mar. 2022).

[5] DEDIS. *Kyber: Advanced crypto library for the Go language*. URL: https://github.com/%20dedis/kyber/ (visited on 15th Mar. 2022).

[6] Ben Adida. 'Helios: Web-based Open-Audit Voting.' In: Jan. 2008, pp. 335–348.

APPENDIX A

INSTALLATION AND TESTING

The D-voting system can be found at `https://github.com/dedis/d-voting`. All instructions for installation and testing can be found in the readme file.