EPFL

DECENTRALIZED SYSTEMS ENGINEERING

CS-438

# Project Report

*Hippocrates: Decentralized, Secure, File Storage*

Group 6
Francesco Intoci, Utku Görkem Ertürk, Aaron Lippeveldts

## 1 Introduction

Hippocrates is a service which allows patients to securely upload and store medical records, which can then be downloaded later by authenticated doctors. Hippocrates relies on Peerster, a decentralized peer-to-peer service we developed during the fall semester for the CS-438 course at EPFL. One of the key features of Peerster is the possibility to share and download files among different, remote peers. It was natural for us to extend this feature, enhancing the power of decentralization: Hippocrates provides a fault tolerant and secure way to share files, with no single point of failure.

## 2 Work Load

- Francesco Intoci: DKG, Cothority, Doctor and Patient APIs (encryption and decryption of files).
- Utku Görkem Ertürk: Blockchain & Paxos, Basic structure of DHT with linear search.
- Aaron Lippeveldts: Blockchain & Paxos, DHT with finger tables and reliability, CLI

## 3 Background

We envision three categories (roles) of nodes in Hippocrates' network:

- Patients, who are the users who upload files (Publish).
- Doctors, who are authenticated users who can have access to such files (Request).
- Cothority Nodes: a collective authority which is formed by a subset of such trusted nodes in the network.

### 3.1 Cothority And Threshold Cryptography: (Francesco Intoci)

The core idea on which Hippocrates' security is founded, is that patients will upload encrypted metadata of files they want to share using the public key of the Cothority which will later relay this information to the doctors who request it. Two problems arise in this system: first, we do not want a single Cothority node to be able to decrypt the information patients share with them. Secondly, it would be disastrous, from a security perspective, to hand doctors the secret key of the Cothority, since this will allow them to decrypt any other file information. Both these problems are solved thanks to a Distributed Key Generation Protocol.

### 3.2 Paxos & Blockchain: (Aaron Lippeveldts & Utku Görkem Ertürk)

For extra security and auditability we added a blockchain to log file accesses. Whenever an authenticated user requests access to a file, the cothority will start a Paxos round to commit the request to a blockchain, together with its timestamp. This would allow the relevant authorities to narrow down who was the cause of data leaks.

### 3.3 Distributed Hash Table (DHT): (Aaron Lippeveldts & Utku Görkem Ertürk)

To make the system scalable and resistant to data losses (reliable) we want to distribute the data into the network using a Distributed Hash Table (DHT). For that we are using the original Chord algorithm. All of the nodes (Cothority, Patients, Doctors) participate in the chord network and store chunks of files. Only Patients can “put” data into the network and only Doctors can “get” data.

## 4 System Architecture

### 4.1 DKG and Secret Sharing: (Francesco Intoci)

As mentioned in Section 2, a great part of Hippocrates' security relies on a Distributed Key Generation protocol. This, in turn, relies on two cryptographic primitives: Secret Sharing and VSS (Verifiable Secret Sharing). Secret Sharing is a cryptographic protocol which allows a subset of *t* nodes to share a secret *k* by distributing shares of the secret among all the *n* participants, where $n \geq t$. In Shamir's Secret Sharing scheme, one node (dealer) will chose the secret *k* by picking a polynomial $f() \in Z_p^*/[X]$ with $p > n$ and degree $t - 1$. The secret k will be $f(0)$ while the shares for a node (participant) $P_i$ will be $f(i)$. The secret can be recovered by collecting $t$ shares and using Lagrange Polynomial Interpolation. [1] One of the problems arising with Shamir's scheme is that there is no way to check whether the dealer or one of the participants are malicious. In order to overcome this issue, VSS is used. A commonly used scheme for VSS is Feldman's protocol [2], which adds to Shamir's protocol an additional step where the dealer publishes a commitment on the coefficient of the polynomial $f()$. Let $k$ be the secret and let $a_j$ be the *j-th* coefficient of the polynomial, for $1 \leq j \leq t - 1$. The commitment will consist of $(c_0, c_1, ..., c_{t-1})$, where $c_j = g^{a_j}$. Hence each participant $P_i$ can check his share $v_i = f(i)$ by checking $g^{v_i} = \prod_{j=0}^{t-1} c_j^{ij}$ . Unfortunately, these kinds of protocols are not a good fit for Hippocrates: having a dealer would imply that there would be a Cothority Node which knows the secret key, thus being able to decrypt every meta data and secret key incoming from a patient. To meet the security requirements we set for Hippocrates, we relied on a Distributed Key Generation Protocol. Essentialy, the DKG protocol will run multiple instances of VSS, where each Cothority node will be at the same time a dealer and a participant (verifier). At the end of the protocol, the secret key will be the sum of all the private secrets of each node, while the public key will be the product of all public keys. The implementation of our DKG protocol follows the protocol described in the paper [3], and heavily relies on the Kyber library developed by the DEDIS Lab of EPFL [4]. The DKG protocol consists in several steps:

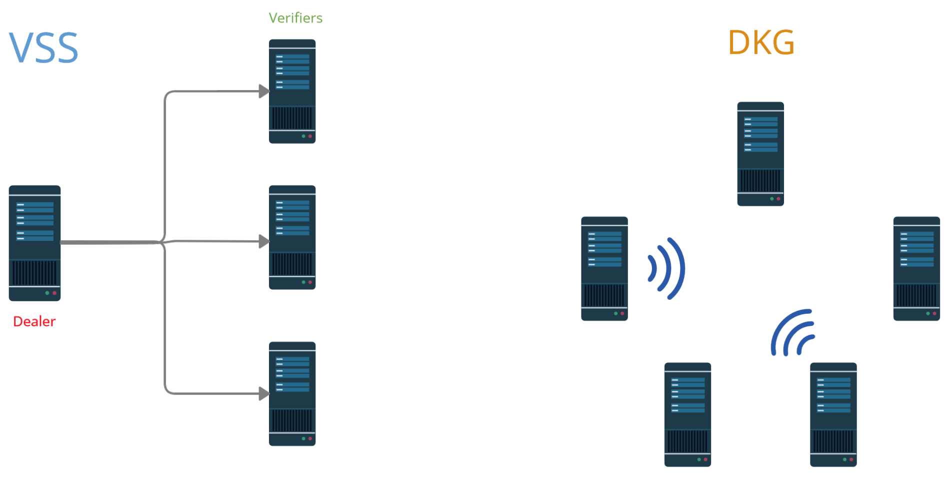


Figure 4.1: VSS vs DKG: each node is a Dealer and a Verifier.

- Public Keys Exchange: Cothority nodes will exchange their public keys for integrity purposes (verification of Deal's signature).
- Deals: Once a node receives the public key of each participant, it will send each participant $P_i$ a Deal, that is the i-th share of this node secret and the public commitment.
- Response: When a node receives a Deal, it will process it and broadcast a Response among all other nodes, to flag whether or not its share was valid.
- EndDkg: After a timeout, based on the Deals and the public Responses received, each node will compute a subset of Cothority Nodes which will be *Qualified*, which in Kyber is addressed as *QUAL* (he will exclude those nodes which are considered invalidated due to communication faults). Based on *QUAL*, a private share of the secret and a common public key will be computed.

Finally, after the set up phase, the Cothority will be ready to serve patient and doctors, through two structures we defined in nodes running as Cothority: *patientAPI* and *doctorAPI*.

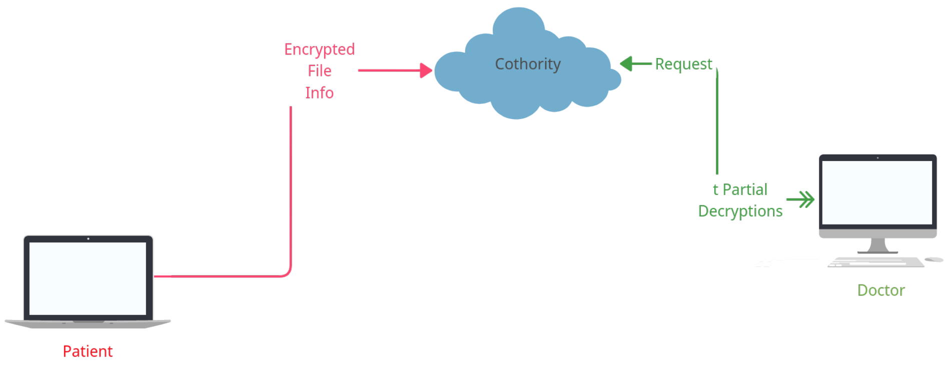


Figure 4.2: Simple schema representing Patient-Cothority-Doctor interaction.

- patientAPI: Patients will split their files in chunks, encrypt them using *AES-CBC-192*, replicate them in several copies, and distribute them in the network. Secondly, he will generate some meta-data (Filename, Owner, Number of Chunks, Copies per chunk) and encrypt them again with *AES*. Finally, he will encrypt the decryption key with the public key of the cothority, and broadcast the encrypted key, the *IV* and the meta-data, to the Cothority.

- doctorAPI: Doctors will ask for meta-data by signing a request with HMAC-SHA256, computed with a symmetric key known by both the doctor node and the Cothority nodes. If the doctor's request is granted, every Cothority node will send a partial decryption of the information using its private share, which will be encrypted with a public key the doctor will provide at request time. The request will also be logged to the blockchain. After collecting $t$ partial decryptions, the doctor can decrypt the information and the patient's key. Note that in this way the doctor will **not** know the shared secret. The cryptographic model used is El Gamal Homomorphic encryption on Elliptic Curves.

For further reading on Kyber DKG implementation, please refer to [5].

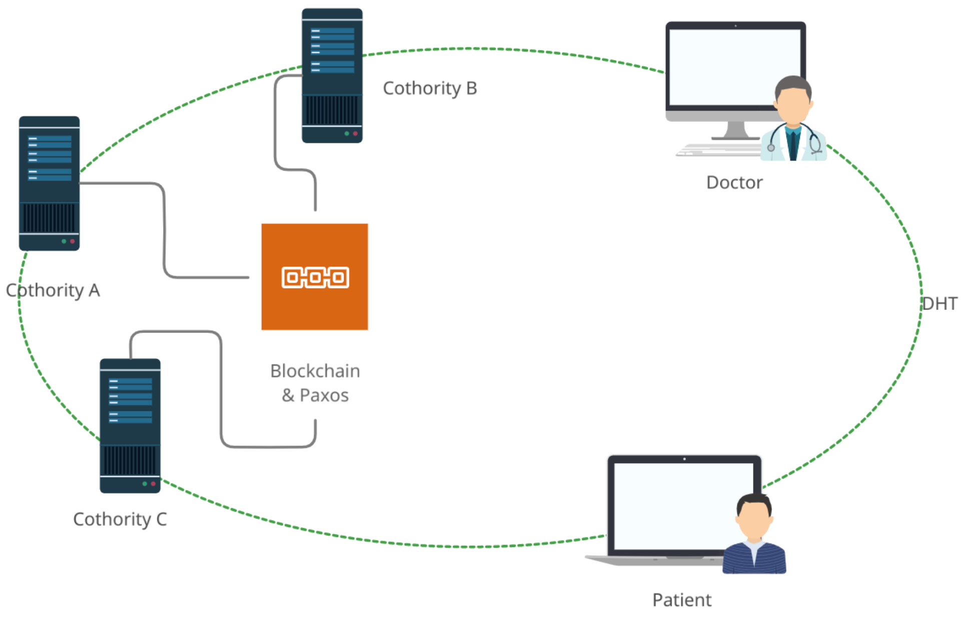


Figure 4.3: System Architecture

## 4.2 Paxos & Blockchain: (Aaron Lippeveldts & Utku Görkem Ertürk)

As mentioned in section 3.2, we use Paxos and Blockchain to log access requests to files. This allows users to see who requested what and when, enabling auditability of the system.

File accesses are initiated when a doctor node reaches out to a cothority node with a file access request. Cothority node will check if the doctor is in the database with a valid private key using HMAC. The node immediately starts a Paxos consensus round for the request. When consensus has been reached, the access is appended to the blockchain. The main benefit of using Paxos here is that all cothority nodes agree on the history. Thus, a compromised cothority node cannot start lying about who accessed what and when.

It is important to note that we are using same rumor algorithm that we used in original Gossiper so all 3 types of the nodes will get Paxos messages through rumor messages but they cannot all interpret it. This is because all communication related to Paxos and the blockchain is encrypted so outsiders cannot tamper with the data or try to submit falsified requests.

## 4.3 Distributed Hash Table (DHT): (Aaron Lippeveldts & Utku Görkem Ertürk)

As previously discussed, we are using a version of the original Chord protocol. We are using SHA-1 to hash keys, ensuring we will get a uniform distribution of the load. As key we are using H(owner name || filename || chunk number || copy number) and for data we are keeping the encrypted version of the chunk. Because the data is encrypted and the filename is hashed, nodes do not know which data they hold. Each node keeps a finger table so that basic hash table operations will be executed in $\mathcal{O}(\log n)$ time, with $n$ the number of nodes. Here, we are assuming that we can approximate the maximum number of the nodes in the network so that each node will have $\log n$ entries in their finger tables. We deal with failures as explained in the Chord paper [6]. Every node keeps 3 redundant successors and when patient nodes want to share their data they "put" 4 more redundant copies in the DHT. So with these two modifications even if nodes leave the network (normally or by crashing), we can both recover the Chord ring structure as well as files up to a point. It is important to note that we are not using Gossiper functions for reliable communication, instead we are using the RPC library with TCP to reach the functions of other nodes. It would not make sense to use rumor messages, for example, as you would then have 2 overlay networks layered on top of each other. This would be bad for performance. In fact, the DHT is mostly independent and nodes can join the DHT without participating in the Gossip network.

Main functions of our DHT:

- FindSuccessor: This function recursively searches the successor of the hash that is given to the function as a parameter. This recursion is provided by RPC calls to remote nodes. It returns the successor of a given key, which is also the node that should hold that key.

- Put: This function first finds the successor of the hash of the key using FindSuccessor and sends an RPC request to the successor to put the data in the local hash table of the successor.
- Get: This function first finds the successor of the hash of the key using FindSuccessor and sends an RPC request to the successor to get the data from the local hash table of the successor.
- Join: Like in the original version of Chord [6], this function first determines which nodes should be in our finger table through the use of FindSuccessor. Then it finds the node who will become our successor and asks it for its predecessor. This node will become our predecessor. We then let our successor know we joined. Our predecessor will find out about our existence through stabilization, so we do not have to notify him. Then we ask our successor if he has keys that we should hold instead and store them in our local hash table if so. We also use FindSuccessor to find nodes which should have us in their finger tables and notify them.
- Stabilize: Occasionally each node will ask its successor for their predecessor. If we should be their predecessor, we notify them and update our successor.

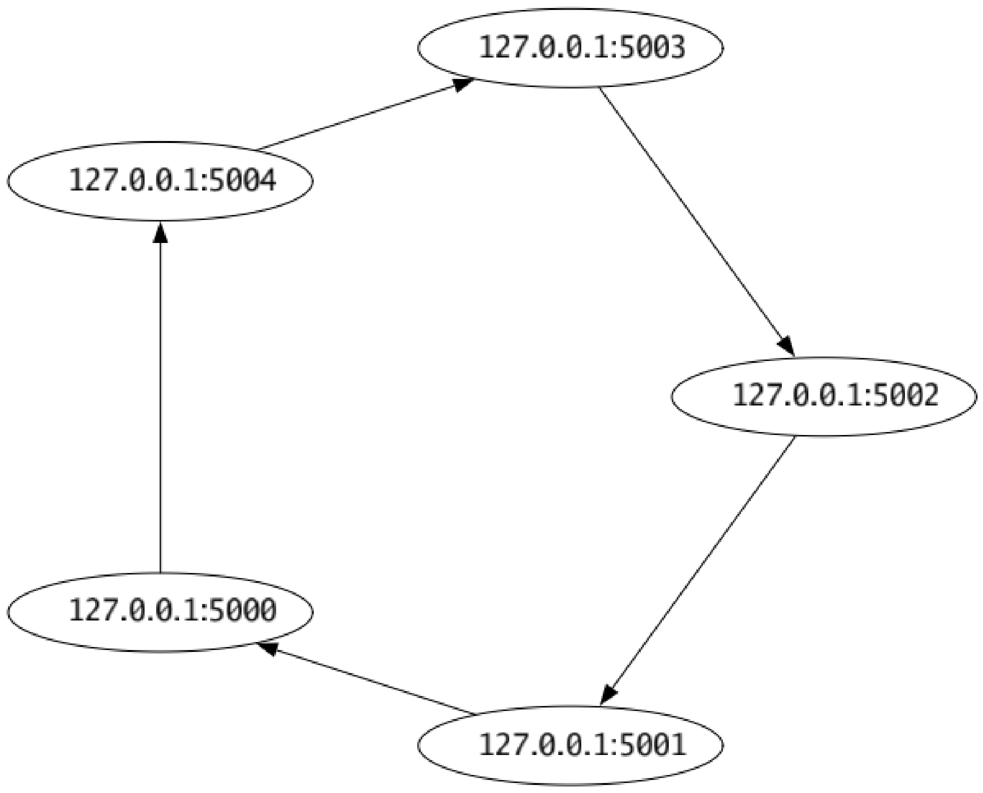
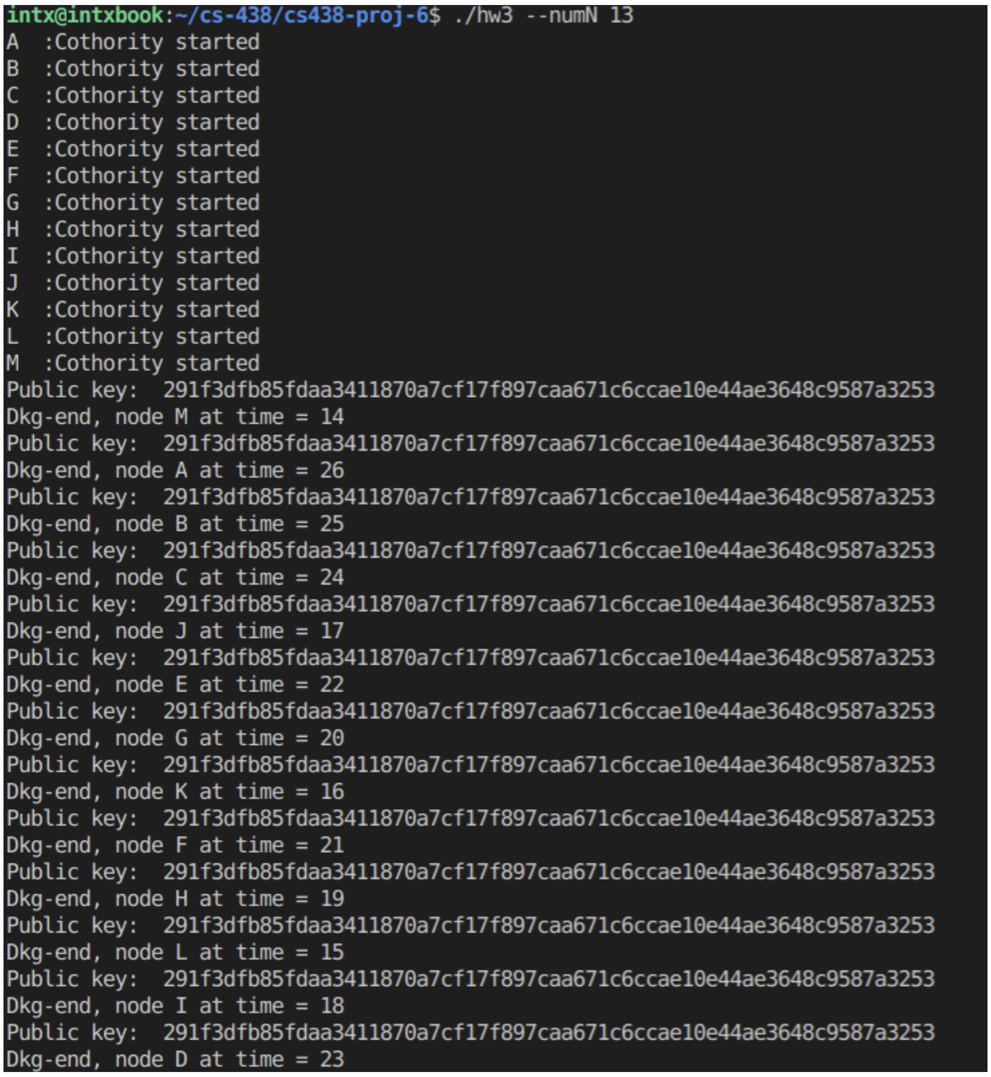


Figure 4.4: Example of a small DHT network

## 5 Evaluation

### 5.1 DKG: (Francesco Intoci)

The biggest challenge we had to overcome when implementing the DKG protocol, was to adapt the Kyber implementation on top of a Peer to Peer service like Hippocrates, which, in turn, is developed on top of Peerster, communicating over an unreliable UDP network.



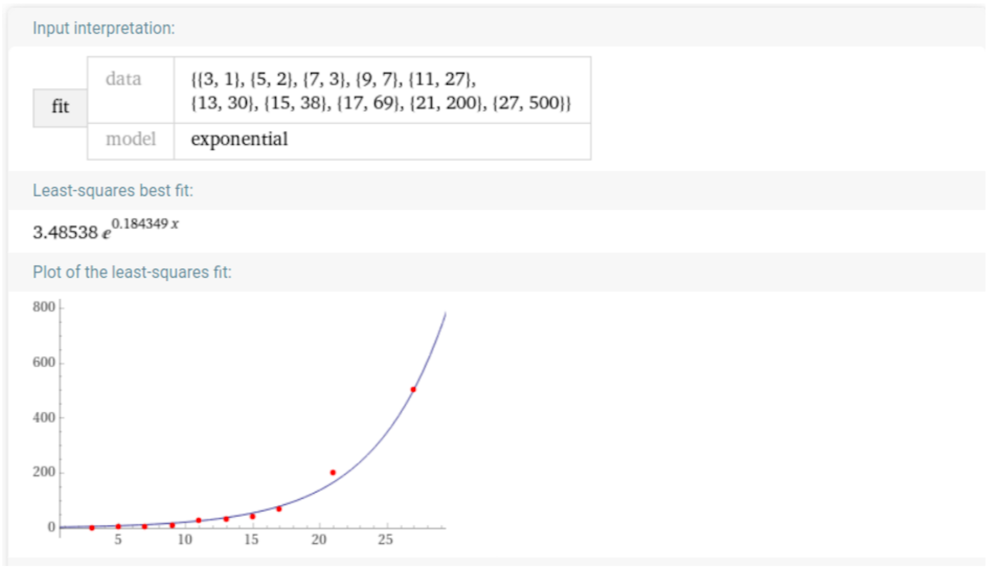
```
intx@intxbook:~/cs-438/cs438-proj-6$ ./hw3 --numN 13
A  :Cothority started
B  :Cothority started
C  :Cothority started
D  :Cothority started
E  :Cothority started
F  :Cothority started
G  :Cothority started
H  :Cothority started
I  :Cothority started
J  :Cothority started
K  :Cothority started
L  :Cothority started
M  :Cothority started
Public key:  291f3dfb85fdaa3411870a7cf17f897caa671c6ccae10e44ae3648c9587a3253
Dkg-end, node M at time = 14
Public key:  291f3dfb85fdaa3411870a7cf17f897caa671c6ccae10e44ae3648c9587a3253
Dkg-end, node A at time = 26
Public key:  291f3dfb85fdaa3411870a7cf17f897caa671c6ccae10e44ae3648c9587a3253
Dkg-end, node B at time = 25
Public key:  291f3dfb85fdaa3411870a7cf17f897caa671c6ccae10e44ae3648c9587a3253
Dkg-end, node C at time = 24
Public key:  291f3dfb85fdaa3411870a7cf17f897caa671c6ccae10e44ae3648c9587a3253
Dkg-end, node J at time = 17
Public key:  291f3dfb85fdaa3411870a7cf17f897caa671c6ccae10e44ae3648c9587a3253
Dkg-end, node E at time = 22
Public key:  291f3dfb85fdaa3411870a7cf17f897caa671c6ccae10e44ae3648c9587a3253
Dkg-end, node G at time = 20
Public key:  291f3dfb85fdaa3411870a7cf17f897caa671c6ccae10e44ae3648c9587a3253
Dkg-end, node K at time = 16
Public key:  291f3dfb85fdaa3411870a7cf17f897caa671c6ccae10e44ae3648c9587a3253
Dkg-end, node F at time = 21
Public key:  291f3dfb85fdaa3411870a7cf17f897caa671c6ccae10e44ae3648c9587a3253
Dkg-end, node H at time = 19
Public key:  291f3dfb85fdaa3411870a7cf17f897caa671c6ccae10e44ae3648c9587a3253
Dkg-end, node L at time = 15
Public key:  291f3dfb85fdaa3411870a7cf17f897caa671c6ccae10e44ae3648c9587a3253
Dkg-end, node I at time = 18
Public key:  291f3dfb85fdaa3411870a7cf17f897caa671c6ccae10e44ae3648c9587a3253
Dkg-end, node D at time = 23
```

Figure 5.1: Final Implementation: test with 13 nodes completed in around 29 seconds

- Naive Implementation: Our first implementation exploited unreliable UDP broadcast in order to exchange Cothority nodes Public Keys and Responses. Even in a simple setting with 3 nodes, this implementation of the Protocol could not end correctly, even when setting timeouts of high magnitude ($t \geq 12$ seconds). Clearly, the reason behind this result was due to communication faults (packet loss), which are common on a UDP network.

- Final Implementation: Taking into account these results, we decided to rely on the Gossip Protocol (Rumor Mongering and Anti-Entropy) we developed for Peerster, in order to enhance the reliability of the broadcasting operation. We managed to instantiate tests with more nodes and lower timeouts, during which the DKG protocol ended correctly. For example, a test involving 13 nodes was successfully completed in around 20 seconds.

The complexity of the DKG protocol, in terms of network traffic, is roughly $O(n^2)$, (where $n$ is the number of Cothority Nodes) since each node will collect $n^2 - 1$ Responses, not taking into account retransmissions. Our solution relying on Rumor Mongering increases the complexity drastically, due to retransmissions. We have thus decided to fit the data with an exponential curve.


Input interpretation:

| fit | data | {{3, 1}, {5, 2}, {7, 3}, {9, 7}, {11, 27}, {13, 30}, {15, 38}, {17, 69}, {21, 200}, {27, 500}} |
|---|---|---|
| | model | exponential |

Least-squares best fit:

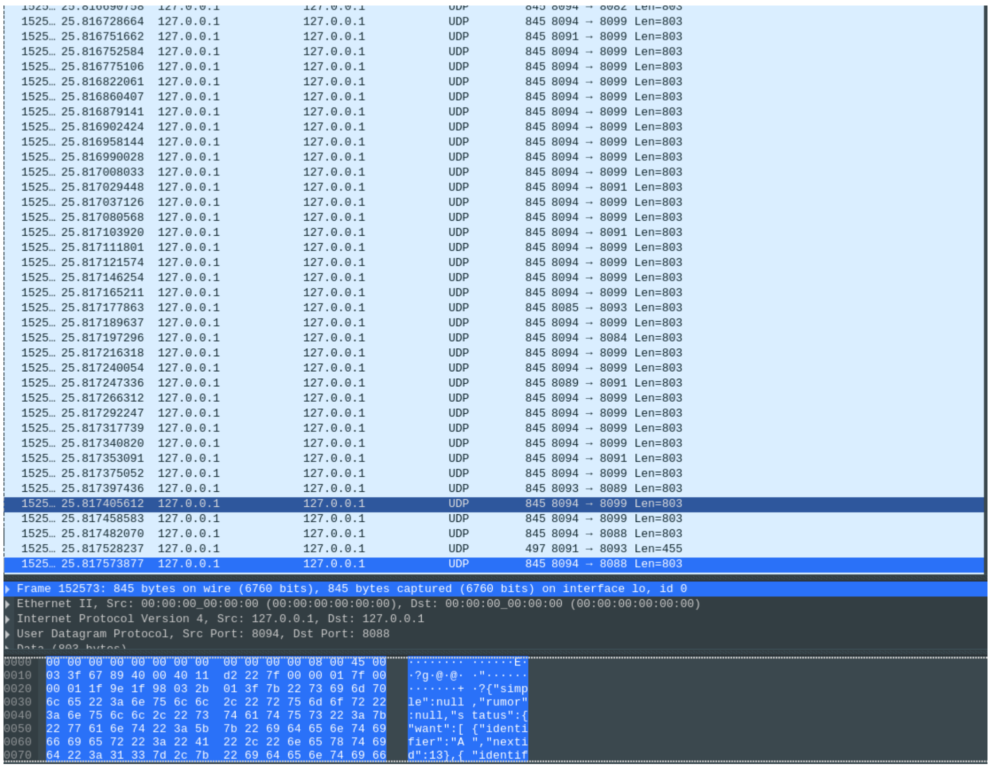
$3.48538\, e^{0.184349\, x}$

Plot of the least-squares fit:

Figure 5.2: Result of exponential interpolation

The resulting plot suggests that the our DKG protocol exploiting Gossiping works well with small-medium size networks. The TTC increases exponentially with the number of nodes. It is worth noticing that we assumed that Cothority nodes are not synchronized, i.e. a Cothority node will not know when the other nodes will have completed the protocol. In order to obtain convergence to the same subset of Qualified nodes, we set the timeout to check the end of DKG at $n.eps$

seconds, where we empirically assigned $eps$ a value, depending on the number of nodes. In conclusion, our solution can be applied only with a limited number of nodes, which could be a realistic assumption taking into account that the protocol is used by a small set of trusted nodes.



```
 1525… 25.816690758  127.0.0.1         127.0.0.1         UDP       845 8094 → 8082 Len=803
 1525… 25.816728664  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.816751662  127.0.0.1         127.0.0.1         UDP       845 8091 → 8099 Len=803
 1525… 25.816752584  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.816775106  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.816822061  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.816860407  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.816879141  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.816902424  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.816958144  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.816990028  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817008033  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817029448  127.0.0.1         127.0.0.1         UDP       845 8094 → 8091 Len=803
 1525… 25.817037126  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817080568  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817103920  127.0.0.1         127.0.0.1         UDP       845 8094 → 8091 Len=803
 1525… 25.817111801  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817121574  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817146254  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817165211  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817177863  127.0.0.1         127.0.0.1         UDP       845 8085 → 8093 Len=803
 1525… 25.817189637  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817197296  127.0.0.1         127.0.0.1         UDP       845 8094 → 8084 Len=803
 1525… 25.817216318  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817240054  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817247336  127.0.0.1         127.0.0.1         UDP       845 8089 → 8091 Len=803
 1525… 25.817266312  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817292247  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817317739  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817340820  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817353091  127.0.0.1         127.0.0.1         UDP       845 8094 → 8091 Len=803
 1525… 25.817375052  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817397436  127.0.0.1         127.0.0.1         UDP       845 8093 → 8089 Len=803
 1525… 25.817405612  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817458583  127.0.0.1         127.0.0.1         UDP       845 8094 → 8099 Len=803
 1525… 25.817482070  127.0.0.1         127.0.0.1         UDP       845 8094 → 8088 Len=803
 1525… 25.817528237  127.0.0.1         127.0.0.1         UDP       497 8091 → 8093 Len=455
 1525… 25.817573877  127.0.0.1         127.0.0.1         UDP       845 8094 → 8088 Len=803

▸ Frame 152573: 845 bytes on wire (6760 bits), 845 bytes captured (6760 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ User Datagram Protocol, Src Port: 8094, Dst Port: 8088
▸ Data (803 bytes)

0000   00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········ ······E·
0010   03 3f 67 89 40 00 40 11  d2 22 7f 00 00 01 7f 00   ·?g·@·@· ·"······
0020   00 01 1f 9e 1f 98 03 2b  01 3f 7b 22 73 69 6d 70   ·······+ ·?{"simp
0030   6c 65 22 3a 6e 75 6c 6c  2c 22 72 75 6d 6f 72 22   le":null ,"rumor"
0040   3a 6e 75 6c 6c 2c 22 73  74 61 74 75 73 22 3a 7b   :null,"s tatus":{
0050   22 77 61 6e 74 22 3a 5b  7b 22 69 64 65 6e 74 69   "want":[ {"identi
0060   66 69 65 72 22 3a 22 41  22 2c 22 6e 65 78 74 69   fier":"A ","nexti
0070   64 22 3a 31 33 7d 2c 7b  22 69 64 65 6e 74 69 66   d":13},{ "identif
```

Figure 5.3: Wireshark capture during test with 21 nodes. Note the frame number and capture time.

## 5.2 Distributed Hash Table (DHT): (Aaron Lippeveldts & Utku Görkem Ertürk)

For the evaluation of the DHT we devised a simple benchmark. We had a node find its immediate predecessor, the node which is the furthest away from it if you only follow successor pointers. We measured the time it took the node to do this 100 times, and took the average. We did this for 2 versions of our DHT.

Our first version of the DHT did not have finger tables, so it was only able to traverse the network by contacting its immediate successor and traversing the ring one node at a time. Our next version did have finger tables and as such was able to take big shortcuts through the ring. We compare the results for the benchmark on both versions in figure 5.4.

As we can see in the figure, with a simple linear search the execution time grows linearly with the number of nodes. This is because for every node that is added to the network, that is another node we have to contact on the way to our destination. On the other hand, the execution time for the benchmark on the version with fingers grows logarithmically, as we would expect for a Chord implementation. The version benchmarked here has 3 fingers (excluding the immediate successor).

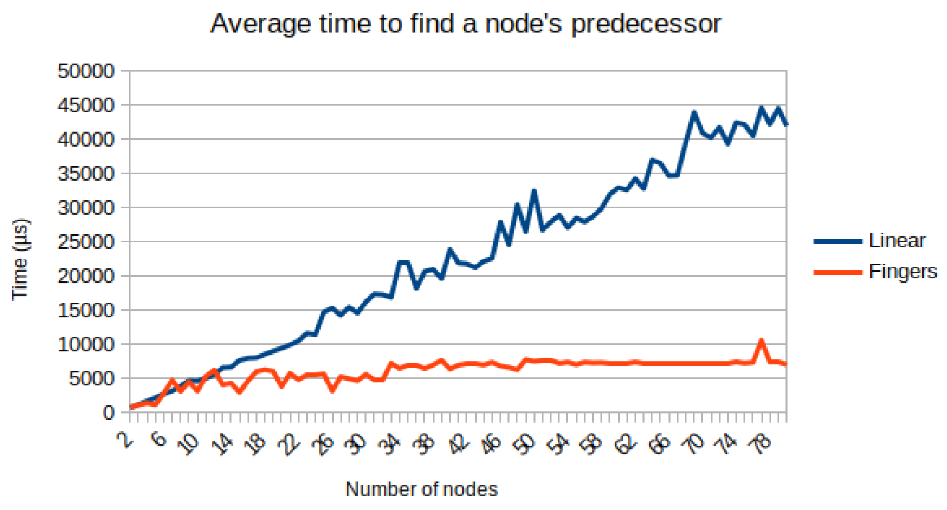


Figure 5.4: Average seeking time of nodes (linear search)

## References

[1] *A.Shamir. How to share a secret. Communications of the ACM.*

[2] *https://en.wikipedia.org/wiki/Verifiable_secret_sharing.*

[3] T.Wong et al. *"Verifiable Secret Redistribution for Threshold Signing Schemes".*

[4] *https://github.com/dedis/kyber/blob/master/examples/dkg_test.go.*

[5] *Improvements to Distributed Key Generation For Use in a Real-World Setting.*

[6] Ion Stoica et al. “Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications”. In: *IEEE/ACM Trans. Netw.* 11.1 (Feb. 2003), pp. 17–32. ISSN: 1063-6692. DOI: 10.1109/TNET.2002.808407.