Decentralized Systems Engineering Fall 2019

Hrusanov Aleksandar, Lanzrein Johan, Rinaldi Vincent

Topic changes w.r.t phase 1 of the project

We decided to still have a way of encrypting data. However to add a twist to the simple key exchange, we design clusters. Each node can decide to become part of a cluster and each cluster allows node to communicate with anonymity and confidentiality. The nodes can either send text messages or call another peer in the cluster with anonymity. Moreover nodes in the cluster can communicate with the group in confidentiality.

Introduction

The motivation behind our project is to have a setting where peers can communicate with anonymity and confidentiality. A practical example of usefulness would be areas where the freedom of speech is not protected and people would not be able to voice their opinion fairly. Having encrypted communication is already a first step to protect their right to express their thoughts. However this is not always sufficient. In some settings, some parties would like it to be impossible to even know if they are communicating. For instance, in a dystopian authoritarian government, where if two parties communicate and one of them is deemed to be a dissident citizen. This government could link all the persons which he has communicated with and flag them as dissidents as well. Even in less extreme cases, users of a chat system should have the right to communicate privately without an eavesdropper knowing who they communicate with.

Security consideration

As we are dealing with private communication we need to take into account adversarial parties. The first adversarial party is the eavesdropper passive adversary. This adversary can listen to communication but can not modify them. If the eavesdropper has access to the whole network or the entire cluster, he can infer where the communication starts however as the peers use network overlay to forward the message and use broadcast it is not possible to know where the message is going. Note that confidentiality of messages can still be preserved as it is not possible to decrypt without the key.

A global eavesdropper can also see what the cluster are and who is a member of each cluster, this is not an issue as any curious node could know be repeatedly query each node to join its cluster until he can map the entire network.

The global eavesdropper is a very strong adversary, a weaker one would be to have one single node eavesdrop on all communications out and in-coming from his peers. This adversary can not learn anything about anonymous communication as there is no clear origin or destination. In a real world setting, it is common to find single or partial network eavesdropper, therefore our implementation would still be able to survive in this scenario.

The other type of adversaries are active. In this case, their toolbox is much more various. A malicious node can flood the network (denial of service), modify messages (tampering), or even withhold messages. In our setting, we do not consider attackers that can do denial of settings as it would require to have a limit of messages sent and a way to identify who the malicious peer is, which is contrary to anonymity. If an attacker modifies the messages sent, for instance encrypted messages, the receiving node will not be able to decrypt it resulting in a failed sending a message. Peerster is using UDP which is an unreliable message delivery therefore we accept that some messages could be modified by nodes and making it impossible to send a message.

About message withholding, in the case of withholding encrypted messages, it will result in a the same as before a failed sent message. However if the malicious nodes joins a network and does not send a heartbeat message or refuses to communicate to help other node generate the shared keys, it does not affect the cluster. As the members can generate a shared key as soon as they are a majority of nodes that agree to communicate, moreover if the node just stays silent he will be removed from the cluster either by voting or by timeout. If a peer that is known for being malicious requests to join a cluster, the peers can vote to refuse it.

In conclusion, our peerster implementation offers resistance to passives partial eavesdropper, and active nodes who will not communicate.

Related work

As we are creating a similar thing to secure messaging, Whatsapp, Telegram, or others come to mind. Those applications rely on the Signal protocol. This protocol allows to have confidentiality and many other desirable security features such as authentication, forward secrecy, message unlinkability and others. However it has one downside as it requires a server to relay the messages and storing the public keys. In our implementation, the relaying is done by peers. Therefore the peerster community is decentralized from any centralized server authorities.

Another protocol which exists, is Bitmessage. This one is much closer to our implementation as it is decentralized and peer to peer. Bitmessage is based on the idea of bitcoin. The user hash their public key and the hash serves as their address. In this setting, the key address pair is trusted as the address authenticates the address. To send a message, peers must compute a proof-of-work. The setting is such that a standard computer takes approximately 4 minutes to send a standard messages. To scale the network, nodes aggregate in a tree and forward messages in the tree.

This approach has the upside that there is no need to trust any central authorities as all messages and keys are decentralized. One obvious downside is the proof-of-work time. Sending a message every 4 minutes is not optimal as peers have a very low throughput of messages.

In our Peerster implementation, peers are able to send messages as soon as they can contact the leaders (Signal protocol analysis : https://eprint.iacr.org/2016/1013.pdf ; White paper of bitmessage : https://bitmessage.org/bitmessage.pdf).

In order to prevent unwanted parties from gaining access to cluster-specific communication, we want to use a session key for encryption/decryption. Group key establishment protocols are generally divided into two groups, one of which depends on a trusted third party, called Key Generation Center (KGC), to generate the key. The second group of such protocols is characterized by the fact that the cluster members themselves generate the key. The latter can be further subdivided into two separate approaches. The ***group key agreement protocol*** has all members of a cluster involved in the computation of a session key (which, typically, results in large overhead cost). The ***group key transfer protocol*** has an *initiator* who initiates the group communication, selects and distributes the key to the other relevant members. A common way to build such a protocol is to employ the secret sharing(SS) scheme (e.g. Shamir's Secret Sharing). In our project, role of the KGC/initiator can be taken by any node in a cluster (e.g. it does not require that the node has any extra knowledge or privileges).

One common way for providing anonymity for a decentralized system is an overlay network. One example is the *Anonymizing Peer-to-Peer Proxy* system (https://mislove.org/publications/AP3-SIGOPSEW.pdf). This approach provides users with the following functionalities ((ii) and (iii) are beyond the scope of our project).

(i) One-way anonymous messages do not contain any information about the origin of the message but has information about intended destination. The sending node encrypts the message with the destination's private key and randomly chooses a neighboring node to send the message to. In turn, this node flips a (possibly weighted) coin whether to send the message directly to the destination peer or to relay it once again. In this way, a non-deterministic path is created from the origin to the destination node, and no node in the path can know if the peer they received the message from is the actual origin or just another relaying node. An anonymity issue not addressed here is that the *destination* of each message is known. This we can solve by adding the following functionality - anywhere along the overlay network, a node can flip a coin to decide between either relaying the packet, or simply broadcasting it to everyone in the network. In that way, only the destination node will be able to successfully decrypt the message.

(ii) Anonymous two-way channels build on top of the one-way anonymous messages described above and have a goal to establish secure request-response channels between nodes. It builds temporary forwarding channels (e.g. path of nodes) between the sending and the receiving node allowing for temporary alternating message exchange. There is a downside to this approach - if a global adversary node exists (e.g. a node that sees the whole network), then he will know that the sender and the receiver nodes are communicating. A possible way to solve this is by simply adding an optional *Origin* field, encrypted within the message.

(iii) Secure Anonymous Pseudonyms provide nodes with the ability to create (potentially multiple) pseudonyms and send messages using them. In that way, receiving node cannot know that two separate messages from different pseudonyms actually came from the same origin. In addition, pseudonym creation depends on a public-private key pair generated by the node itself, thus, does not depend on a central authority.

The goal of our project is to provide a messaging and/or audio streaming service similar to Whatsapp or Skype, those applications being known to implement end-to-end encryption. In opposition to Whatsapp which is a centralized application, in the sense that it is managed by a central authority, Skype is however actually decentralized, as the infrastructure on which is based our Peerster application. *The network contains three types of entities: supernodes, ordinary nodes, and the login server. Each client maintains a host cache with the IP address and port numbers of reachable supernodes. The Skype user directory is decentralized and distributed among the supernodes in the network* (https://worddisk.com/wiki/Skype_protocol/). *Supernodes relay communications on behalf of two other clients, both of which are behind firewalls or "one-to-many" network address translation. Without relaying by the supernodes, two clients with firewall or NAT difficulties would be unable to make or receive calls from one another* (https://en.wikipedia.org/wiki/Skype_protocol).

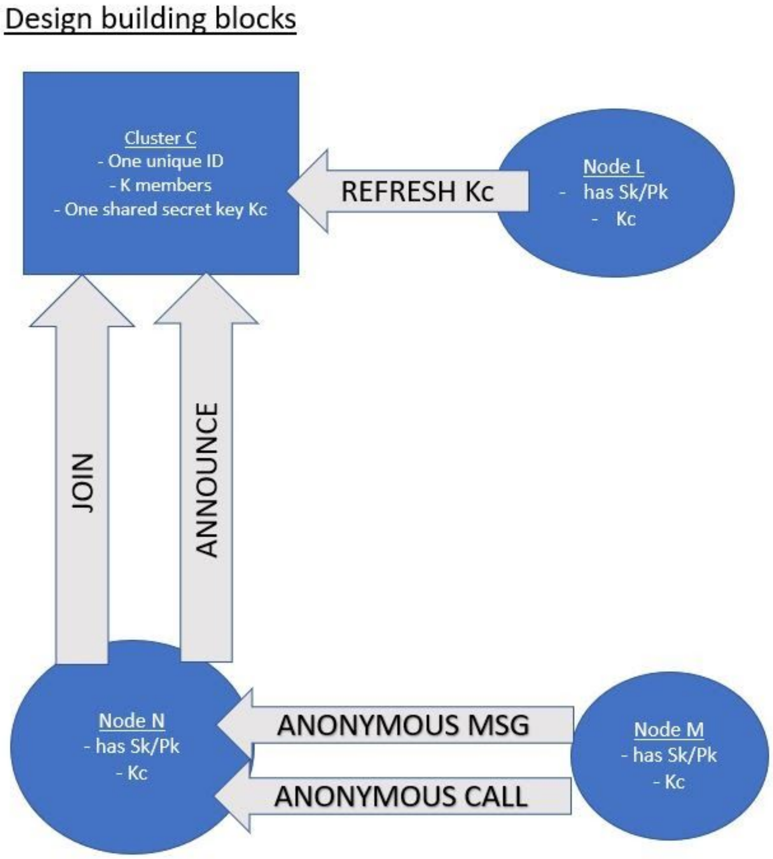
Our application gets its inspiration from the Skype protocol : we decided to create clusters of nodes but without any kind of predefined supernodes to avoid single point of failure. Each node acts as a common user, member of a single cluster, that may receive requests from external nodes, send requests concerning changes on the cluster community he is into, or be randomly chosen to perform shared key computation and distribution. We also want the cluster size to be reasonably limited to reduce the complexity of the network, since that, first of all, by keeping a relatively small number of nodes inside one cluster, we enhance the reachability process by trying to obtain an upper-bound of hops as low as possible, but secondly, each node will not have to carry a huge amount of work because of the restrained number of other node members of the cluster it has to deal with. This reduces the public radius of a cluster network, which makes it a bit more private than Skype or Whatsapp, but the motivation of this choice is to enhance privacy and security, by limiting the number of potential malicious external nodes to enter the cluster, eavesdrop on the communications, and leave the cluster to leak the retrieved private information to other clusters.

This members limitation particularity comes in addition to the encryption of messages exchanged through the cluster’s paths, a simplified voting protocol for cluster’s community decision (as the expel of a given node judged to be malicious for example) and the provisions taken to ensure that each node can communicate inside only one cluster, like encryption/decryption keys resets and cluster members updates after a certain period. This renders the protocol secure against impersonation and man-in-the-middle attacks, which have been shown to be feasible (to some extent) attacks in some messaging systems.

System goals, functionalities and architecture

Cluster

We define clusters as a set of nodes. Each node can only belong to one cluster at a time. On startup, a node belongs to a cluster of its own. A peer *A* can **request** to join a cluster by sending a **JOIN** message to a known peer *B* (a member of the cluster-to-join).
Once the request has been received, the cluster can accept or refuse the peer. When accepting a peer *A*, the cluster contact point B and *A* proceed to generate a shared key using the **Integrated Encryption Scheme**. This key is then used to exchange the cluster secret key, which is used for confidential group messaging. B will also send the public keys of the members in the cluster and broadcast A public key.



Each node is expected to store the keys of members of the cluster. As can be expected in networks, some peers will disconnect or want to join another cluster. For this case,the peers are expected to send a **HEARTBEAT** message to ensure that they still want to participate in the cluster and are still alive. On a hard timeout, set to 5 minutes, the keys will start a key rollout which will eject unresponsive nodes.
This design choice has the advantage of solving two issues : when a peer wants to leave a cluster, he does not have to announce his request to leave as he can just stop sending **HEARTBEAT** messages and, in that way, will be removed, and the cluster will not fill up with dead members.

Creating a new cluster

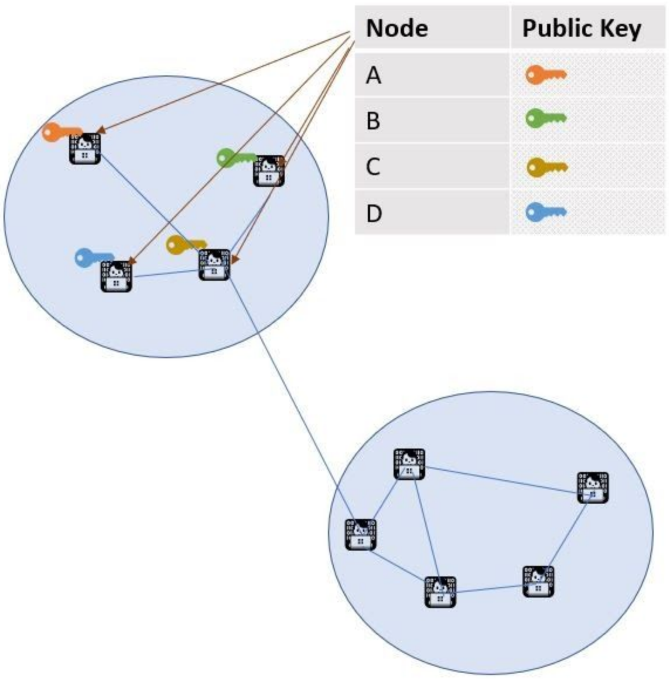
Any peer can decide to create a new cluster. When doing so, he will automatically become sole member and generate a random 64-bit ID for his cluster. According to the scale of our application, this makes the chance of a collision negligeable.
He will then wait for other peers to request to join his cluster. When starting a cluster, a peer can decide to have a limit of members that can join it, either for stability of his own local machine or to make it more scalable.

Key rollout

On a set timeout, the peers in the cluster will initiate a key rollout. This is to ensure that no dead nodes stay in the cluster and that node who left can not continue to interact with the cluster. They will select a random node L that will generate a new shared key K_c. Each node will generate a new keypair Pk, Sk and send to the designed node L. Which will then send to every node in the cluster all the new public keys and the new shared key.

Encryption schemes

In our project there will be more than one level of encryption. There needs to be an encryption for cluster wide announcement, as well as for confidential messages. We also need to have a protocol to handle cluster membership. They will be described in the following.



Node	Public Key
A	🔑
B	🔑
C	🔑
D	🔑

Architecture of clusters

Cluster membership

When asking to join a cluster *C*, a peer *A* will contact a peer *B* belonging to the cluster. The peer *B* will ask his cluster if they accept A explained below. If the membership is accepted, he will send the parameters of the cluster and the shared key K_c to *A*. *A* generates a fresh private and public key pair pk_A and sk_A and sends broadcasts it to the cluster.
Both parties compute a shared secret key using a key derivation function (in our setting we will use *hkdf* from the package https://godoc.org/golang.org/x/crypto/hkdf). *B* then sends the secret key sk_C, which is used for cluster-wide messaging, and stores the pair (A, pk_A) in a table. He also forwards the table of public key and members of the cluster.

Cluster anonymous messaging

Once a peer belongs in a cluster he can communicate with anonymity with any node in the same cluster. Formally, A wants to send an anonymous message to B. Given Pk_B and a message to transmit m. A computes $c = Enc(Pk_B, m)$ and sends it to a random peer. The peer flips a coin and either sends it to another random peer or starts broadcasting the message. If a peer receives a broadcast message, he verifies if the hop limit has been reached. If yes he stops broadcasting it, otherwise he broadcasts it as well. Then he tries to decrypt it if he succeeds then its for him with a very high chance.

This can also be done in a similar fashion for audio streams or files. There might be a drop in quality and a lag.

Cluster-wide messaging

The cluster key is obtained once joining the cluster or if there is a key rollout. Once this key is obtained, peers in the cluster can encrypt a message m under the key K_c and only peers in the cluster can decrypt the resulting ciphertext.

It is forwarded in a broadcast fashion if a peer receives a ClusterMessage. Once receiving a message the peer tries to decrypt it and forwards it.

Decentralized E-voting protocol

The reason why we wanted our application to use the concept of clusters of nodes, is to strengthen and guarantee privacy and security through our network. First of all, using clusters of limited nodes is useful to reduce the potential number of malicious attackers that can enter and interfere in the clusters communications at the same time. On another side, anonymity through communication helps to protect the privacy of the users inside a cluster, the cryptographic encryption techniques used enhance confidentiality, whereas integrity and authenticity are highlighted thanks to the generation of public key/secret key pairs, and shared keys, for each member. The fact that the activity of the nodes that are not part of a given cluster is limited helps to prevent malicious external nodes that may listen to internal communications in order to break confidentiality and/or tamper the messages sent through the cluster, then leave the cluster to join another one, and leak the retrieved information to other nodes from other clusters. Concerning the concepts of anonymity and clusters members management implementation methods, they have been explained previously (encryption of the sender's personal information and message for anonymity, and conditions on the acceptance of a new node requesting to join the cluster for cluster participants management). Note that provisions are also taken in addition to guarantee that each node can communicate inside only one cluster, such that periodic cluster (and so, members) information updates and key pairs/shared keys resets, initiated by a randomly chosen node from the cluster (also explained previously).

Since no single node can take arbitrary radical decision for the whole cluster because of the fact that we don't want the existence of any supernodes (to keep our implementation the most decentralized as possible), a simplified "E-voting protocol" is used in the case a node broadcasts a request which would affect the cluster's community, such as the exclusion of a member suspected to be malicious, or the acceptance of a new member that previously requested to this member node to join the cluster. An E-voting instance can be generated from any node as a request with a specific flag (expel a given node, accept a new member, ect...) that is broadcast to every other node of the cluster. After the 5 minutes hard timeout mentioned previously in the "Cluster" part, *(n/2) + 1* nodes will be randomly selected as new "Authorities" to count the votes (such that *n* is the number of nodes in the cluster, and this number of "Authorities" obtained is rounded to the smallest closest integer). Each node broadcasts their response ("yes" or "no" that can be also represented as integers like "1" and "-1" respectively) to every "Authorities" (an "Authority" node only broadcasts its response to the other "Authorities" since broadcasting to itself is useless). If an "Authority" still doesn't receive a number of responses equal to the number of nodes in the cluster after a certain time, it broadcasts the same voting request that has been sent from the node that first initiated the voting procedure. After a certain predefined specific time that we can call as "voting time limit" which is synchronized through every node in the clusters, the "Authorities" broadcast their voting results to every node in the cluster, and compare their results with other "Authorities" nodes. If there is a difference between one given pair of voting results (how many "yes" and how many "no") then the votes are cancelled, and the same number of new "Authorities" nodes are randomly chosen, those nodes which will send the corresponding voting request, and the voting procedure will repeat, until we reach the valid voting final result. This voting comparison procedure, by selecting a majority of nodes to check the votes, can prevent malicious nodes that may want to tamper with the votes and modify them. The final official results of the voting procedure is the answer that as been selected in majority. In case of equality (if we have an even number of voters) we take the result that as been selected in majority by the "Authorities".

Audio Call system

We propose the following architecture for an anonymous call. Between two node A and B. First A needs to contact B with a **CALL** message. B can either **ACCEPT** or **DECLINE.** Once the call has been accepted, A and B both start to send encrypted "GossipPackets" with data. The data is the voice data. Once arrived the data is decrypted and output as sound on each A and B machine. To preserve anonymity, the packets would still follow the same overlay network concept as for normal messages. This makes it harder for an observer to infer if there is a call or if there is just a lot of messages.

Task distribution

Aleksandar

- Cluster Anonymous message handling
- Anonymous calls

Johan

- Key generation, encryption and decryption of packets
- Key rollout of clusters - needs testing
- GUI

Vincent

- E-voting protocol for node members acceptance/exclusion

Implementation

Encryption (Johan Lanzrein)

The encryption scheme can be found in the *ies* package of the project. We first focus on how to encrypt packet using a symmetric key. For this purpose, we use methods *Encrypt* and *Decrypt* in the *encrypt.go* file. We make use of AES encryption scheme with CBC encryption mode. The cryptographic operations of AES with CBC mode are done by the *crypto* package of the standard library. One trick in this part was to take into account the for the padding introduced by the block cipher. At first I did not consider it but using unit testing I was able to recognize the issue before it became too big and take it into account.
Afterwards, we focus on how to derive the shared key for two parties communicating. Instead of using already existing libraries, we implement it on our own. The first question is how to define the keys. We are going to be using the ECIES so we decide to have keys of 32 bytes size. We use the curve “Ed25519” and make use of *curve25519* to derive the public key once the secret key has been sampled. Once the public and secret key are generated, a party can derive a shared secret by using the *KeyDerivation* method which given a key pair and a public key will derive the shared secret. The derivation is done by using *hkdf* and a combination of both secret and public keys. In the section result, we discuss the speed of the encryption and decryption for different data size.

Clustering (Johan Lanzrein)

The clustering needed to have a few elementary methods : Initialization, Joining, Leaving and key rollout. All of these methods have been implemented in the file *clustering.go*

Initialization

Initalization is done when a gossiper wants to create its own cluster. It will start a new cluster with only itself as a member. Then it will start the heartbeat routine that sends updates to all members of the cluster.

Joining

Joining is done by creating a *RequestMessage* inside a *GossipPacket*. The gossiper needs to know the ID of the cluster and the name of an other gossiper in that cluster to be able to join. Once he has those parameters, the gossiper can create a request to join which will then be received by the other gossiper. The receiver then starts the e-voting protocol to decide if the requester is accepted. In the end, the requester gets a reply if he was accepted or not in the cluster. If he is accepted, the replies contains the metadata of the cluster.

One challenge here was to have synchronization accross all new members. Indeed if a gossiper A asks to join a cluster consisting of B,C,D,E. And B,C,D vote to accept him, E needs to be made aware that there is a new member. For this we use the key rollout, once there is a key rollout, the newest members are then added to the metadata of the cluster for everyone.

Leaving

If a member wishes to leave the cluster, there is a method *LeaveCluster*. This method sends a broadcast to all members announcing his wish to leave. Moreover, the method stops the heartbeat routine and removes all data about the cluster.

Key Rollout

This part of the clustering was the hardest one. The selection for the orchestrator of the key rollout is done by using a PRNG to select the member. The PRNG is shared across the cluster to ensure that all members **.** Afterwards, once a rollout happens, each member of the cluster will send its newly generated key to the orchestrator. The orchestrator collects the new keys, generates a new master key and broadcast the new cluster metadata encrypted with the previous master key. The rollout also allows to learn about new member that may have been missed !

Broadcast messaging

The broadcasting is done by having a new type of packet *BroadcastMessage*. Which will contain the ID of the cluster and the data encrypted with the master key of the cluster. The broadcast message is sent to all members of the cluster who will decrypt it and output it. Besides that these type of message are useful to perform some other routines. In the case of the heartbeat, it is a *BroadcastMessage* with an empty text field. If the gossiper would like to leave the cluster, he can use this type of message and make use of the *leave* flag to request to leave. Finally, these message are used to send information about the cluster in case of a key rollout.

Anonymous messaging and calls (Aleksandar Hrusanov)

Anonymous Messaging

The protocol uses the cluster members as an overlay network, creating a non-deterministic path between Origin node and Destination node. Every intermediate node, decides on receiving an anonymous message not destined for them whether to relay the message through a random cluster member or to route it to the destination.

From the original Peerster functionality, the anonymous messaging support private anonymous messages (e.g. sending text messages). Every cluster member has information about other members' public keys. If node A wants to send an anonymous message to node B, he creates a GossipPacket with a Private message in it, encrypts the gossip packet ($c = Enc(Pk_B, m)$), and sends a GossipPacket with an Anonymous Message.

The Anonymous Message structure holds the encrypted data *c,* a plain text Receiver string (needed for the relaying nodes if they decide to route the message and for the destination node to know the message is for him), a RelayRate (between 0 and 1), and a RouteToReceiver boolean. The RelayRate is specified by the sender and is used by the intermediate nodes as a weighted coin - the higher the value is, the higher the chance that each intermediate node will relay the message is. In other words, with increased value of RelayRate, the path between origin and destination nodes is expected to increase in length, thus, increasing the anonymity of communication.

The RouteToReceiver boolean is set to true by the node who, after flipping a coin, decides not to relay the message anymore. In that way, every consecutive node will now **not** to relay the message, but to keep on routing the message to the destination.

Lastly, a sender can choose between **full anonymity** and **partial anonymity**. If *full anonymity* is chosen, the encrypted message will **not** contain any information about the origin. This means that even the receiving node will not know who sent the message to him. This presents a trade-off between full anonymity and communication capabilities, as in this setting, full anonymity does not allow for request-response type of communication. (Note: as stated in the proposal, we reviewed a solution for that, *anonymous channels*, but it is beyond the scope of the project). The solution which we implemented supports *partial anonymity* - besides the origin node, the destination node will also know who sent the message (e.g. the origin's name will be part of the encrypted data). This makes it possible to adapt the anonymous messaging functionality to support other Peerster message types such as DataRequest and DataResponse, FileSearch, etc. and any request-response protocol, in general.

Calls

The calling functionality uses a request-response model and relies on a few main message types.

CallRequest

If a node A wants to call a node B, he sends a CallRequest message (simply containing Origin and Destination strings). On receiving, if B is not in another call already, he decides whether or not to pick up. If he is currently in a call with another node, then an automated call response is sent informing A.

CallResponse

The CallResponse message holds information about the *status* of the call as decided by the receiving node B. The status can be:

- *Accepted* (meaning node B picked up), in which case the audio streaming is initiated between nodes A and B;
- *Declined* (meaning node B declined the call), in which case no audio streaming is initiated, states of the two nodes remain unchanged
- *Busy* (meaning node B is currently in another call), in which case no audio streaming is initiated, states of the two node remain unchanged

HangUp

The HangUp message is used in two different situations. If nodes A and B are in a call, either party can send a HangUp message to the other, the audio streaming ends for both and the states of the two nodes changes so it reflects that they are not in a call anymore. If node A is calling node B and B is not picking up, node A will send a HangUp message to B after 10 seconds of waiting. In that way, B will know that A is no longer calling.

AudioMessage

Once a node A has called node B and node B has accepted, audio streaming is initiated between the two nodes. During this time the two nodes exchange AudioMessages which contain encoded raw audio data. Golang wrapper for the PulseAudio library is used to record raw Pulse-code Modulation data. Golang wrapper for the Opus codec library is used to encode the recorded raw data before sending it over an AudioMessage and decode it on receiving by the other node. Then, PulseAudio is used to playback the decoded raw audio data.

Call Anonymity

The call functionality is an example of a request-response anonymous messaging. All of the call-related messages described above are sent via the overlay network explained in the Anonymous Messaging part. The only difference is that we do not give the user the option for full anonymity (because of the request-response nature of the communication) and also have a constant RelayRate of 0.5 (for larger clusters, higher RelayRate can result in latency in the audio communication).

E-voting protocol (Vincent Rinaldi)

The E-voting protocol is used to allow some vetting on the new members that want to join the cluster. It also allows to remove members in case they are too disruptive. The E-voting protocol is done implicitly when a new member requests to join the cluster. Otherwise it can be triggered by requesting to ban a member.

Protocol

The protocol works as follows : suppose a party “C” would like to join the cluster of “A”. First of all, “C” needs to know of the existence of A to be able to proceed. “C” will then send to “A” what a called a *JoinRequest* or an *ExpelRequest.* “A” will then add this request to its own list of pending E-voting requests and will broadcast the request to the rest of the cluster.

Each node belonging to the cluster that receives the broadcast *Join* or *Expel* message will display a notification to tell the client that an answer is required, which will display the identifier of the request in the GUI in a text box under the form “JOIN C” or “EXPEL C”. In addition, if the receiving node is an *authority*, he will add the request identifier to a two-dimensional slice of of string (*[][]string*) to gather the votes of every cluster members (it will be added to the first index of a “row” in the 2D string slice).

Every client will then have to provide an answer to the voting item. Only two possible answers can be selected : “ACCEPT” (meaning that the member is in favor of the request) or “DENY” (if the member is against it).

The response of each member will be broadcast and only *authorities* will react to it : when they receive a vote (*Accept* or *Deny* message), they will append it to the corresponding “row” of the two dimensional string slice (the corresponding row is the one having the related request identifier on the first index) under the form « 1 : NODE D » or « 0 : NODE D » where NODE D is the concerned voter (1 means that he voted ACCEPT and 0 is for DENY).

When an authority received a number of different votes for a given request equal to the number of members in the cluster (or number of members minus one if the request is an EXPEL type, since the node that is concerned by this EXPEL request can’t vote), it broadcasts a *Compare* message to notice the other *authority nodes* that he is ready to share and compare its results.

When it receives a number of *Compare* messages equal to the number of *authority nodes* existing in the cluster (or number of *authority nodes* minus one if the node to EXPEL is an *Authority* itself), the *Authorities* broadcast theirs results, which is the “row” of the slice related to the concerned request, under a *Results* message.

When an authorities receives a slice of string types representing the votes gathered by an *Authority*, it will use a map of the type *map[string][]string* with the following map scheme : (name of corresponding request) → (list of Authorities from which it received a slice of votes which matches its own results for this request). If an *Authority* receives a list of votes that doesn’t match his own list, then the votes are cancelled, and a *Cancellation* message is broadcast. The process of the “Votes Cancellation” is explained at the end of the section.

When receiving the same number of valid (matching) slices results (or *Results* messages) than the number of *Compare* message needed previously (number of *authority* or number of *authority nodes* minus one according to the situation), the *Authority* will count the votes, and, after doing it, will broadcast the official final result of the votes for the concerned request under a *Decision* message. In case of an equality, the *Authority* will then count the votes again, but only those of the *Authorities*. Since there is always an odd number of *Authorities* strictly greater than half of the number of members (explained more in detail at the end of the section), a majority decision will be always reach in the case of a JOIN request, or EXPEL request concerning a non-*Authority* node. When the EXPEL request concerns an *Authority*, then in case of strict equality between every member votes but also only between *Authorities*, then the request final decision will be DENY (this means we only execute the request when a strict majority of nodes ACCEPT the request). When receiving *Decision* message, the node deletes in its maps and 2D slice the entry concerning the request.

Concerning the node having stored physically the pending request at the beginning (which was Node A in the example at the beginning of the section), when receiving the *Decision* message, it will be the only one to react to this broadcast *Decision* message by replying/indicating the resulting decision to the node concerned by the request, with a *Reply* Message (he will also remove the pending request from its pending requests list (so it won't react anymore to a subsequent broadcast *Decision* message concerning the same request). If a request is accepted, the cluster members will be updated and the cluster will be transposed to a new state at the next *Key Rollout*. Every slices and map used during the protocol will be reset after *KeyRollout*.

Concerning the Cancellation of a request, at the moment an *Authority* receives a *Results* message containing a slice of votes that doesn't match its own list of votes, the node will broadcast a *Cancellation* message. When the node, having stored physically the concerned pending request at the beginning,receives this *Cancellation* message, it will broadcast a *Reset* message. Every node inside the Cluster will react to the *Reset* message by broadcasting a *AckResend* message, to indicate that they are ready to provide again a vote for this request. When the node containing the pending request receives exactly a number of *AckResend* messages equal to the number of members in the Cluster, the *Join* or *Expel* request can be broadcast again through a *Join* or *Expel* message, and the protocol concerning this request starts again. Obviously, the 2D slice and map entries related to the request are deleted when receiving a *Cancellation* message and before broadcasting the *Reset* message, so the protocol can restart at the same state it was when broadcasting the *Join* or *Expel* request the first time.

This concludes the protocol.

Implementation

Four client command line flags have been added to initiate a JOIN/EXPEL request and to vote ACCEPT/DENY for them : *-joinOther ; -expelOther ; -accept ; -deny* that all take a string type as argument. This argument is the node concerned by the request for the flags *-expelOther, -accept and -deny*, and is one of the nodes belonging to the cluster we want to join for *-joinOther*. This means *-joinOther* can only be called by a node not belonging to a cluster, and *-expelOther, -accept and -deny* can only be called by nodes belonging to the cluster where the request is broadcast. The argument of *-expelOther* must be a node belonging to the same cluster as the client node initiating the EXPEL request.

Concerning the structures, we added a broadcast message type, having different boolean attributes (*JoinRequest, ExpelRequest, AcceptProposition, DenyProposition, CaseCompare, ResultsValidation, FinalDecision, CancelRequest, ResetIndictation, AckResend*) in order to specify the type of broadcast messages during the E-voting protocol. Those types of messages are set accordingly in the *evoting.go* files which gathers all the functions that broadcast those types of messages in the whole cluster according to the current step of the E-voting protocol for a given request.

We also added a *Results* attribute to the *RumorMessage* structure since we are encapsulating *RumorMessages* inside *BroadcastMessages* to communicate the elements needed to achieve the voting protocol for a given request. On the client side, the *Message* structure has now a *PropAccept, PropDeny, JoinOther* and *expelOther* attribute which have been added to store the argument of the client flags that we described previously.

The *clustering.go* file contains the main functions concerning the E-voting protocol, the Cluster update, and the operations on Cluster members, like the *ReceiveBroadcast* one which handle the received *BroadcastMessages* and performs the correct operations according to the boolean field that has been set to true depending at which step in the e-voting process the request is, the functions *ReceiveDecisionJoinRequest* and *ReceiveDecisionExpelRequest* that are used to reply or communicate the final community decision to the node concerned by the request, and the different *ClusterUpdate* function for the addition of a node after accepting a JOIN request, or the expulsion of a node in the case the Cluster community decided to accept an EXPEL request, or the *Key Rollout* process to transfer the Cluster to an updated state.

Concerning the choice of adding *Authorities*, since a malicious node is able to modify a list of votes, we wanted to select a bunch of nodes that would count the votes and compare them between each other *Authorities* list of votes. We only proceed to the decision of the request if every *Authority* has the same list of results. This reduces greatly the probability to decide after counting votes that have been tampered. If one list of results doesn’t match an other one, the whole E-voting process for this request must restart.

The number of *Authorities* is always Odd and strictly greater than half of the number of cluster members. Their election as *Authorities* is completely random. There are two situations when new *Authorities* are selected : when a *Key Rollout* occurs, and when a request must be reset after an *Authority Node* receive a list of results that doesn’t match its own.

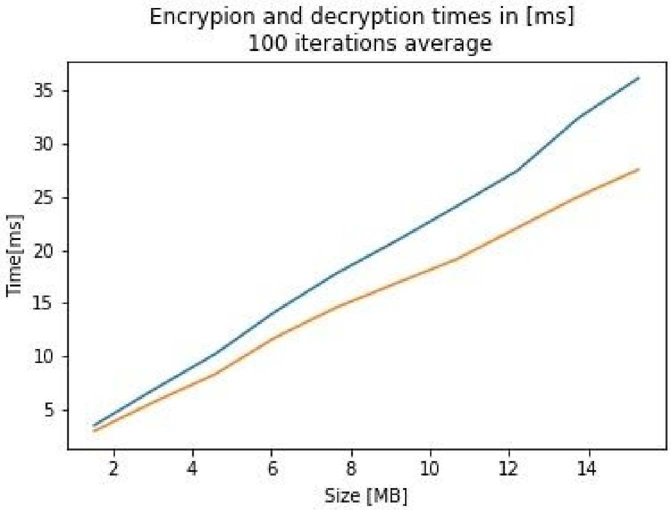
The main challenge to implement our E-voting protocol was to find a way to adapt and include the random selection of *Authorities* into the *Key Rollout* process and try to synchronize correctly each node to receive the same information about the new state of the cluster, since the number of members can constantly change between two periodic *Key Rollouts*. Also, deciding on the voting rules, when there is an equality after counting every Cluster members votes, or especially when an *Authority* was excluded from the voting process during the procedure concerning an EXPEL request of which he is the target, making the number of voting *Authorities* even, had to be correctly and properly handled in a convenient way.

Results

Encryption computation

We benchmark the computation time of the encryption and decryption process.
Moreover we have computed the time taken for the key generation and key derivation however they are very little (< 1 ms) so it was not possible to benchmark it accurately.
In the figure below, we can see that for very big packets (~15 MB) the execution time is very acceptable.

Afterwards the execution time scales linearly as expected. In Peerster, the packet size is very small, usually less than a few KB, meaning this encryption scheme is acceptable for our purposes and will not be a bottleneck for the service. The result of our benchmarking can be seen in the figure below.

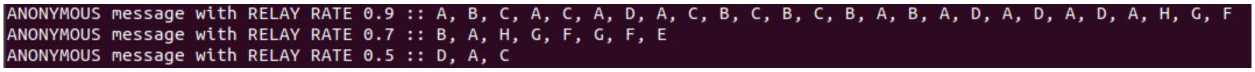


GUI

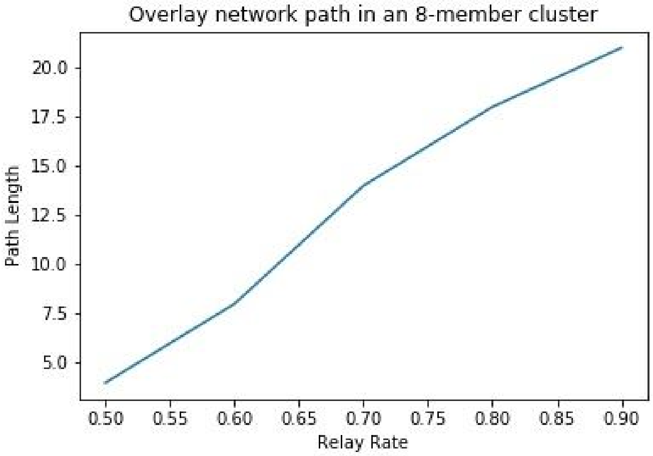
The GUI consists mainly in a panel that will display all the information relative to the cluster of the current gossiper. In this panel there are options to create a new cluster, join an existing one or to leave it. Also a gossiper can send an anonymous message or initiate a call by clicking on the corresponding symbols.

Anonymous Messages

We ran some tests to measure the path length which an anonymous message takes from the sender to the receiver via the overlay network. The tests were performed in a 8-member cluster with a relay rate of 0.5 (from A to F), 0.7 (from B to E), and 0.9 (from D to C). The results below indicate the number of relaying nodes the anonymous message went through before being routed to the destination node.



```
ANONYMOUS message with RELAY RATE 0.9 :: A, B, C, A, C, A, D, A, C, B, C, B, C, B, A, B, A, D, A, D, A, D, A, H, G, F
ANONYMOUS message with RELAY RATE 0.7 :: B, A, H, G, F, G, F, E
ANONYMOUS message with RELAY RATE 0.5 :: D, A, C
```

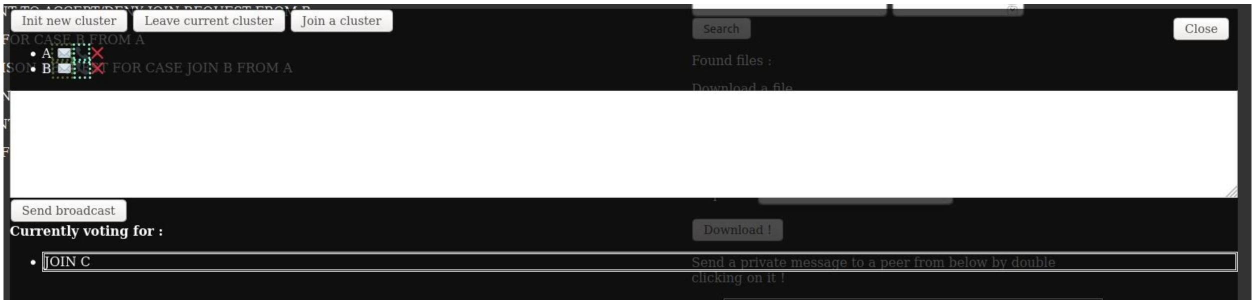


Calls

The call functionality shows good performance in terms of record/stream/playback speed but suffers from a slight echo effect*. Golang wrapper for Opus uses CGO and executes C code under the hood which in turn can cause problems when dealing with concurrency. We switched from PortAudio (Golang wrapper using CGO) with PulseAudio (not using CGO) for recording and playing PCM data, but relying on Opus for encoding/decoding of raw PCM data could potentially cause issues related to concurrency as we cannot fully rest on the safety of Go anymore.

*Note: The echo effect is reinforced in the demo because it was recorded on one machine (e.g. both peers were recording and playing back the same sound simultaneously).

GUI panels and views

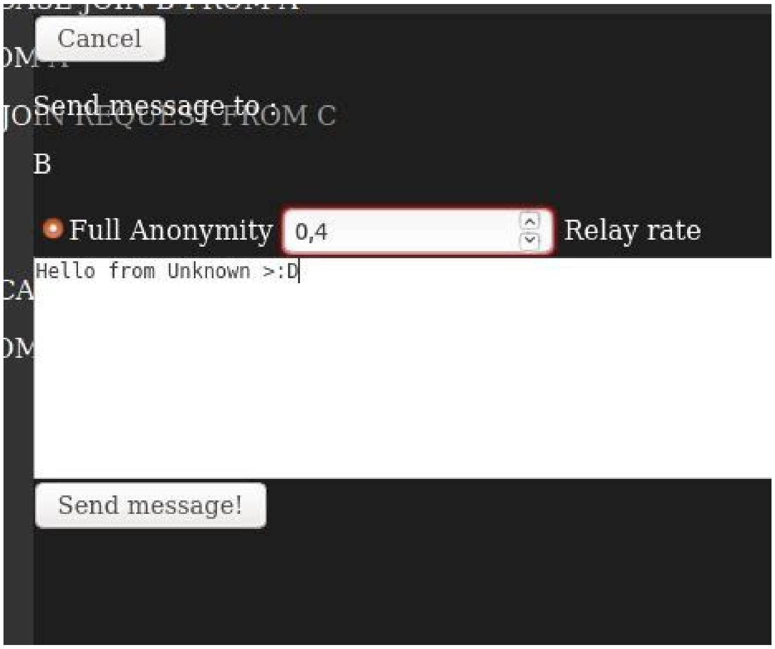


Currently voting for :

- JOIN C

Send a private message to a peer from below by double clicking on it !

Main panel



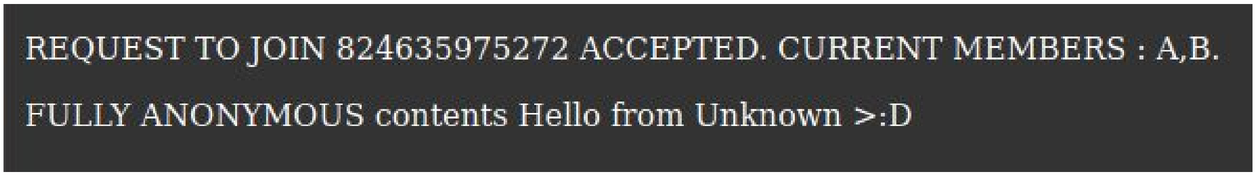
Send message to :

B

● Full Anonymity 0,4 Relay rate

```
Hello from Unknown >:D
```

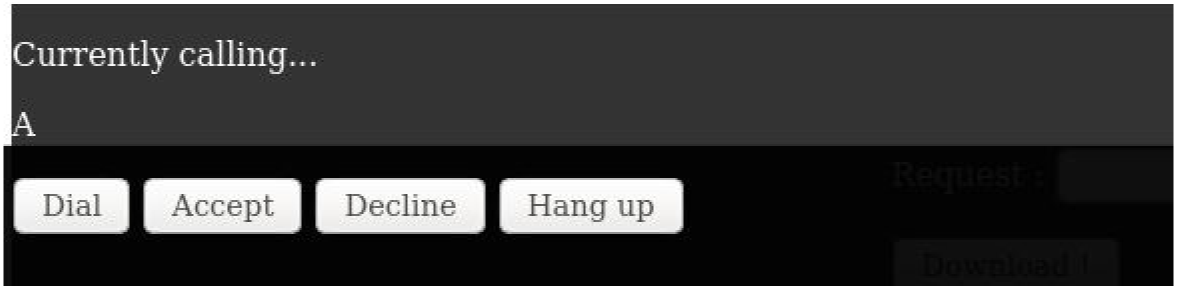
Anonymous messaging panel



REQUEST TO JOIN 824635975272 ACCEPTED. CURRENT MEMBERS : A,B.

FULLY ANONYMOUS contents Hello from Unknown >:D

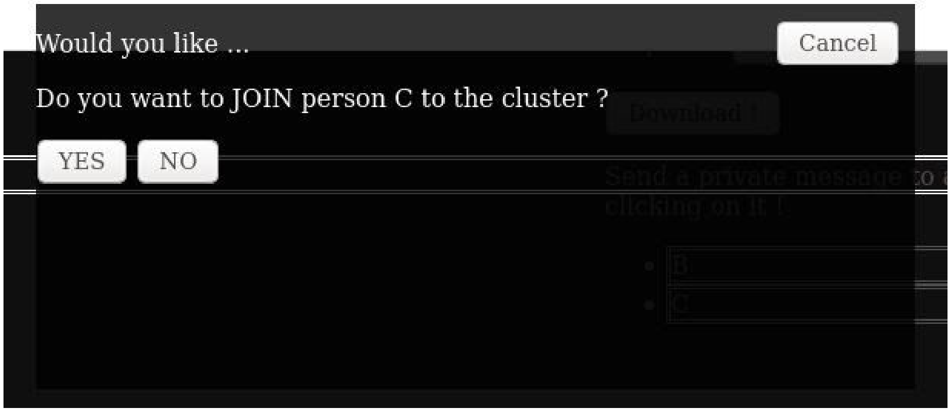
Result of an anonymous message in B's GUI



Currently calling...

A

Call panel of node B, calling/in a call with/incoming call from node A



Would you like ...

Do you want to JOIN person C to the cluster ?

Voting panel

Conclusion

This project allows to introduce some anonymity and confidentiality in the messaging of peerster. Moreover the e-voting allows to have communities with cluster that can self regulate with a democratic mechanism. With some encryption scheme such as ECIES, we can introduce confidentiality to the messages. Moreover, with the help of the communities in the form of clusters, we have a way to communicate public keys between other trusted peers. The master key enables group communications. On top of text messaging, we have added a calling functionality that allows to call someone else with anonymity.

Finally, the clusters are self regulated as there is a voting protocol to allow newer members to enter or to expel existing ones, this allows to have a decentralized self-regulated community.

In future work, we could implement the existing standard Peerster functionalities such as file sharing with encryption or file search inside a community. We could also optimize the e-voting to have to option to vote on more items than joining or expelling.