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).

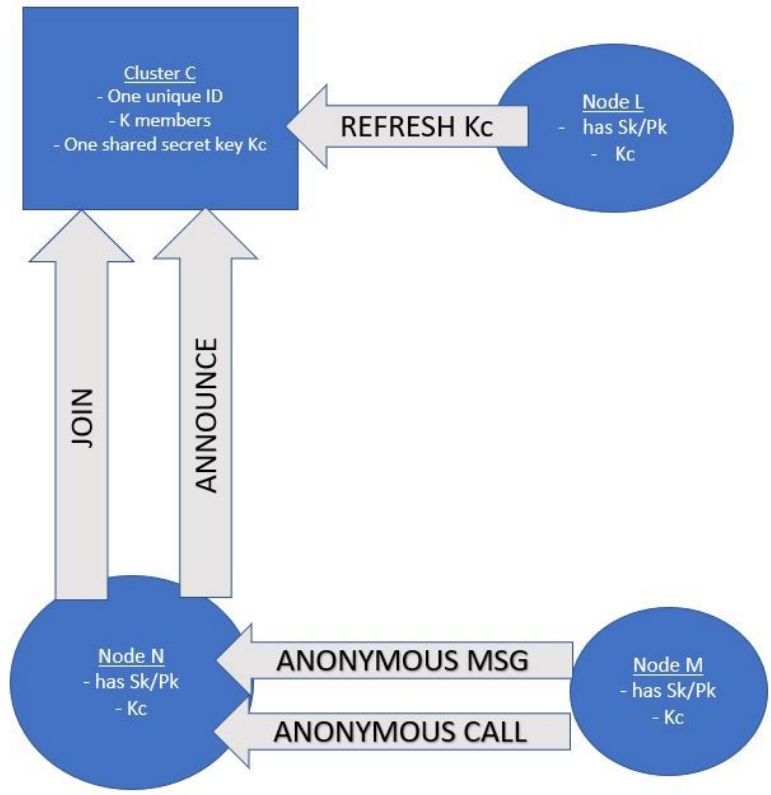
Our application gets its inspiration from the Skype protocol : we decided to create clusters of nodes but without any kind of predefined supernodes to avoid single point of failure. Each node acts as a common user, member of a single cluster, that may receive requests from external nodes, send requests 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

Design building blocks



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

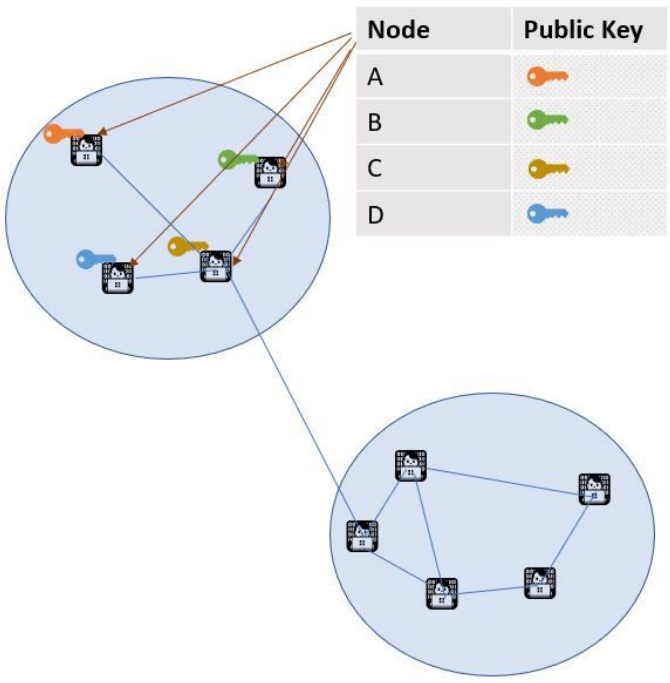
Any peer can decide to create a new cluster. When doing so, he will automatically become sole member and generate a random 64-bit ID for his cluster. According to the scale of our application, this makes the chance of a collision negligeable.
He will then wait for other peers to request to join his cluster. 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.



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
- GUI - initiate/receive anonymous messages, initiate/receive/end anonymous calls

Johan

- Key generation, encryption and decryption of packets
- Key rollout of clusters
- Cluster generation and merging (includes GUI showing of what cluster user is in)

Vincent

- E-voting protocol for node members acceptance/exclusion
- GUI view of voting