P2P Chat

Melnikov Ignat
Uglovskij Artem

March 2021

1 Introduction

Decentralized systems are getting more and more popular nowadays. We tried to create an app for communication and sharing files without a host. Also we prevent a security issues which can lead your message being cracked. In the next sections we will explain how it works now and what can be done to make our application better.

2 Current protocol

When user runs our app for the first time, he is to come up with a login and password for further launches and remember them. Then it takes some time to generate new pair of keys: private and public. When it is done, the application save these keys in a key storage to extract them next times.

Before starting main activity we also update (or add new line if it is the first entrance) our table of users with owner data: ip address, public key and status. Then the main activity is started.

Primarily, the TCP[6] and Multicast[7] receivers are launched to listen and analyse requests from other users. Then interrogator process begins to send *hello-messages* to all members of our network. When we have collected the information about others, we can begin to communicate with them by sending and receiving text messages (we do not have the ability to send files yet).

3 Network

First of all, we should define columns in table, which contains the information about users, and in tables, which store dialogue contents with each user. From now on we will call them *users' table* and *dialogue table*.

id	name	ip address	public key	AES key	status
1	Artem	192.168.42.1	MIICI1..	AB2K..	online
2	Ignat	192.168.42.10	MIICI6..	ABF8..	offline
...	...	...	...	...	...

Table 1: Users' table

id	author	timestamp	message
1	Artem	17346558	hello, Ignat
2	Ignat	17346590	hi!
...	...	...	...

Table 2: Dialogue table

3.1 Multicast receiver

The multicast[7][13] receiver is responsible for checking the status of the network as well as adding new users to the table. It receives advertising requests and, depending on the type and public key of the recipient, decides what to do next. So, there are 3 possible situations:

1. *recipient's public key* is „all“
2. *recipient's public key* is our public key
3. *recipient's public key* is not our public key

In the first case we should check either *sender's public key* is our *public key* or not. If it is not, then we attempt to find this key in *users' table* and if we are not succeed then new line with information about this user is added to the table. Otherwise, if such user was found then we update his *ip address*, which we can get from the message with an appropriate java-api method. In both, we must send a response to the sender informing him that we are online. The response message is shown below. And if *sender's public key* contains our *public key* then we simply ignore this request as it is ours.

recipient's public key	sender's name	sender's public key

Figure 1: Response message format

In the second case we should either add a new user in our table or update some fields for an existing one. This type of request means that we have got a response for our *hello-message* sent by *interrogator*. So, if such *public key* is already in our table, we renew *ip address* and set *status* to *„online“*.

In the third case we should ignore this message because it is not addressed to us.

It is worth noting, that we search in the *table* exclusively by *public keys* as only they guarantee the uniqueness of the user.

3.2 Message (TCP) receiver

TCP[6][12] receiver listens on the specific port for incoming dialogue messages from all users, who have our public key and ip address in their users' tables. The format of such messages is given below.

sender's public key	encrypted message	type

Figure 2: Dialogue message format

We need *sender's public*[5] *key* to find the matching *AES*[3] *key* in our *users' table* in order to decrypt the content. *Type* determines whether we received a regular message or an *AES key*. As will be described in the *Cryptography* section, the dialogue between two users is encrypted using symmetric key. But first, we must somehow transfer this key. Dialogue initiator, while trying to cipher the message, finds out that there is no *AES key* for this user. So, he generates such key and sends it first, then he sends the message encrypted by this key.

Thus, we can decipher the information, store it in *dialogue table* and notify the application to show the message to user.

3.3 Message sender

In order to send massages we use Message sender. It is responsible for ciphering, connecting and dispatching data from one user to another.

Sending a message starts by checking the recipient's status in *users' table*. If it is „offline“, then we notify the user about it and do not take any further actions. If the recipient is „online“, then we try to establish the connection with him. If within a certain time (2 seconds) it was not possible to connect, then we notify the user about a connection-error and change the recipient's status to „offline“. After the connection is established, we attempt to retrieve the symmetric key defined for the given recipient from the table. If the attempt was unsuccessful, then we generate a key, encrypt and send it, after which we transfer an original encrypted message as we already described earlier in *Message receiver* section. After all successful actions a relative row is added in *dialogue table* and user can see sent message on his screen.

3.4 Interrogator

This process must periodically send *hello-messages* after specified amount of time so that *Multicast receiver* gets responses and update *users' table*.

Before sending requests interrogator sets *statuses* of all users in out table to „offline“. Thus, after *Multicast receiver* has processed all the responses, we will have only online users in out table.

To avoid simultaneous mailing we should define random timeout. In our case it is $(3000 + rand\_time)\, ms$, where $rand\_time$ is randomly chosen from $-500\, ms$ to $500\, ms$.

The content of *hello-message* is shown below.

all	our name	our public key

Figure 3: Hello-message

4 Cryptography

To protect the personal data of users, the most reliable algorithms to date, tested over the years, were used.

4.1 Asymmetric

As it was already mentioned, when authorizing a user for the first time, the new pair of keys is generated. This is done using the RSA[5] algorithm, 4096 bits long. This key length makes it impossible to crack, and also allows you to easily encrypt symmetric AES[3] keys, which will be discussed later. Key pair of this size is generated for a significant amount of time, however, the generation of such a pair is a one-time event for each account. Also, the public key is used as a unique user identifier, similar to how it is done in popular cryptocurrencies. This is done for the convenience of transferring your account between devices, if the MAC address was chosen as the user identifier, many problems would arise.

4.2 Symmetry

However, asymmetric keys are not suitable for encrypting the message stream and, in the long term, files, as they require processing power and significant time. To reduce the delay when sending and receiving a message, symmetric AES-256 keys are used. They are generated, exchanged and saved automatically when you try to send the first message (see *Message receiver* for more details).

It is also a great advantage that the hash obtained by the SHA-256[2] algorithm can be used as a secret key. This allows us to keep the private key in an encrypted state by simply encrypting it with a hash from the user's password.

4.3 Message authentication code(MAC)

However, there is another problem, any third party user can send spam messages on behalf of another user. To solve it, MAC calculation is used using a common AES key by the HMAC[4] algorithm. Then it is added to the block of each message and upon receipt it is recalculated and checked, if there is no match, the message is read as spam and ignored.

SealedObject: msg	MAC msg

Figure 4: Encrypted message with MAC

4.4 Counter

The system is still susceptible to duplicate message spam attacks. Let the user Bob write a message to Alice, and the evil Tom listens to them and saves some messages.

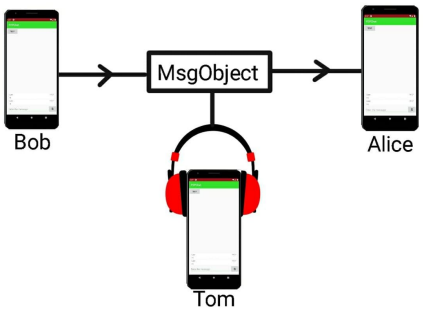

Figure 5: Evil Tom

At any time, Tom can send Alice a copy of the old message, and this copy will be validated and displayed. To avoid this, Alice and Bob must keep a chat message counter and check that the counters are equal each time a message is received. It doesn't make sense to pass the counter value itself for validation, since Alice knows which counter to expect. Instead, we will calculate MAC using a common symmetric key and add its value to the message.

SealedObject: msg	MAC msg	MAC counter

Figure 6: Encrypted message with counter

4.5 Some additions to the implementation

1. An important part of the implementation is the moment when symmetric keys are exchanged, since it is not entirely clear how the MAC is calculated in this case.

   First, the hash of Alice's (recipient's) public key is calculated, which is used as the AES key to calculate the MAC of the transmitted key.

   $$MACmsg = HMAC(AESconnectionkey, SHA(AlicePublicKey))$$

   This way anyone, including Bob and Alice, can figure out the MAC key. However, this will not help Tom in any way, because the AES connection key can only be decrypted with Alice's private key, but at the same time, the presence of the MAC saves Alice from spam messages with false keys.

2. The second important point is how the private key is stored in the permanent memory of the phone.

   To securely store the private key, it is encrypted using the AES key generated from the user's password in a unique way. And the file is created in such a way and with such rights that Android as a system protects access to it, prohibiting other applications from reading and editing it.

5 Future

During rallies in Hong Kong, the government turned off the internet in the city center, thus depriving them of communication and a way to organize. Then the Fire Chat application came to the rescue, it used the modern apple framework (Multipeer connectivity framework)[9], which adds the ability to connect devices to the mesh network.

More than 7 years have passed since that moment, Wi-Fi and Bluetooth technologies have become not only faster, but also, which is important for mobile devices, less energy expended. For 7 years, 7 versions of Android and IOS have been released, wireless communication protocols have been added or updated. Also, before our eyes, the popularity of decentralized systems (such as blockchain[10]) is growing, they are actively developing and many works on this topic are being published.

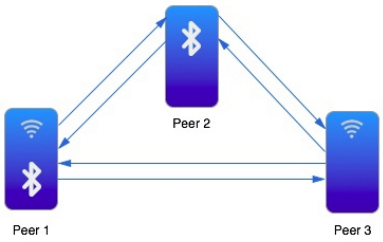

Figure 7: Multipeer connectivity framework

In the future, it is possible to make not only a decentralized chat, but also use all devices as a distributed data storage system, in the same way as BitTorent and IPFS[11] do. You can also add a system for storing and confirming messages, like DAG networks[8] (an alternative technology to the blockchain, in which each user can write their messages to the network without using the services of miners or validators).

We may be living in a fairy tale, but such networks in the long run can cover entire cities and create decentralized social networks and databases.

6 References

1. Tomas Falcato Costa, Peer-to-peer Communication in Android Devices: Web Access over an Ad Hoc Network, 2017

2. Federal Information Processing Standards (FIPS) Publication 180-1, Secure Hash Standard (SHS), U.S. DoC/NIST, April 17, 1995.

3. Federal Information Processing Standards Publication 197, Advanced Encryption Standard(AES), 2001

4. Federal Information Processing Standards Publications 198-1, The Keyed-Hash Message Authentication Code (HMAC), 2008

5. RSA, https://ru.wikipedia.org/wiki/RSA

6. Darpa Internet Program Protocol Specification, Transmission Control Protocol, 1981

7. Cisco, IP Multicast Technology Overview, 2001

8. Qin Wang, Jiangshan Yu, Shiping Chen, Yang Xilang, SoK:Diving into DAG-based Blockchain Systems

9. Apple, Multipeer Connectivity
https://developer.apple.com/documentation/multipeerconnectivity

10. Satoshi Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System, 2009

11. Juan Benet, IPFS - Content Addressed, Versioned, P2P File System,

12. Android, Socket
https://developer.android.com/reference/java/net/Socket

13. Android, Multicast Socket
https://developer.android.com/reference/java/net/MulticastSocket

14. Android, p2p connections via Wi-Fi Direct
https://developer.android.com/training/connect-devices-wirelessly/wifi-direct