

ALGORITHMS FOR PEER-TO-PEER MESSAGING SYSTEM

Problem

In the area of instant messaging and VoIP, there are currently a bunch of solutions, built with fairly similar architecture.

For messaging, the industry-standard is to use so-called client-server architecture, when each client (where client means a piece of software, run on client device - smartphone, desktop computer, laptop, etc.) communicates with server (where server means a piece of software, commonly known as back-end, which runs on a computer, usually owned by instant messaging service provider, and accessible over Internet). When a client A wants to send a message to another client B, he sends it to the server with mark «message for client B», and client B then can pull this message from server (pulling can be done periodically, or can be triggered by some events).

For VoIP calls, the scheme is generally the same. The difference is, that some of VoIP clients use so-called peer-to-peer connections when transferring voice/video signals, because using server to re-translate the signal is usually too expensive in terms of resources (the server internet connection is usually fast and broad-band, but still limited, it's computational powers are usually significantly larger, than on any client device, but also limited). Peer-to-peer connection is a networking connection when two devices connected «directly» in terms of networking.

There are two huge problems in these schemes. First, this solution's infrastructure has «critical» points - e.g. disabling server means disabling the system overall. It can be done intentionally (hacker attack) or accidentally (disaster, long-term power failure, etc.). Even if there is more than one server, their number is still limited, and in multi-server architecture there usually exists a DB server or servers, whose number is even less, than the number of servers, that communicate with clients.

Second, each active client (e.g. client who communicated with server at least once) uses computational resources and internet channel of the server, meaning each client costs some money to a company owning a server.

This application is intended to eliminate or significantly decrease the effect of the above 2 issues.

State of the art

There are well-known solutions, that operate in so-called «server-less» mode. In the area of IT, more than a dozen of years exist a bunch of P2P file sharing services (like BitTorrent or e-Mule), that are almost server-less. The creators of this protocols did not create servers at all or reduced their functionality so that the mentioned price of single active user is significantly lower, compared to traditional architectures.

When instant messaging over IP became more and more popular, these technologies were brought to messaging. Widely-known solutions are Bleep, Tox and many others.

These messengers used peer-to-peer (direct) connections for messaging (peer-to-peer was used only for VoIP before).

What it actually means - for these messengers to operate they do not need servers. Each client in their network is itself a small server.

They operate well and solve the problems of traditional architecture, but only when we speak about laptop and desktop computers.

Though most of these messengers have mobile client, they are quite not usable, because smartphones operate differently, than the desktop or laptop because of energy saving policy. Smartphones do not allow applications to operate constantly (for example, iOS gives an application a small period of operation time when it is putted into background and then this app will not have any processor time until it is pushed forward), or warn user about it (like on modern Android devices, the user is informed if any application consumes too much battery power). Still, even if constant operation is allowed and user is not warned, it is not good for application to drain battery too fast - it's not practical for users and they will switch to another solutions.

What it all really means, from one side we want clients to communicate directly with each other, from the other side, we understand, that mobile clients will (and really should) most of the time stay offline (and their mobile application will be unavailable for direct communication).

So, with smartphones specialty in mind, we come to a problem of offline messages. This problem does not exist in traditional architecture (server is supposed to be online «eternally»).

Another problem is, even if we create any solution for offline messaging, what should we do with security. Current security trend and industry-standard is so-called «perfect forward security». It means, that:

- 1) each message should be encrypted;
- 2) encryption key should be negotiated frequently;
- 3) if client's secret data (long-term key used for client authentication) were stolen, it must not affect the security of already-sent data.

Here we should put a side note about long-term keys. Each client has a so-called keypair - a Public Key and a Secret Key, generated locally when user registers in messaging system or when he first signs-in. This keypair used in authentication operations. The property of this keys is that when some message M is encrypted with one key, it can be decrypted only with the other one. So, if we encrypt something with Secret Key (SK), the notation is $SK(M)$, to get M back we have to make $PK(SK(M))$. PK is accessible by everyone (that's why it is Public). This keypair, again, is used only for authorisation, and this scenario is called «signing».

So, if SK becomes known to a hacker, we must ensure that previously sent messages can not be decrypted. To do so, we can use Diffie-Hellman algorithm, or any of it's derivations, but it is still needed, that both users communicate, and, in peer-to-peer scenario, require that both users have to be online when one desires to chat with another.

These 2 problems are commonly known as «Asynchronous Offline Messaging» and «Asynchronous Perfect Forward Security», and this application addresses both of them.

Invention description

In this application we propose the messaging system, based on peer-to-peer communications, that resolves 3 general problems:

- 1) We have to beat the problem of large per-user cost of traditional architecture;
- 2) We have to deal with offline messages;
- 3) We have to provide perfect forward security when one user is offline.

The first problem is resolved partly with peer-to-peer communication scheme, but still exists even in P2P. We must note here, that not all client devices, specifically - not all internet connection types of client devices, allows peer-to-peer

communication. Majority of clients connected to Internet via so-called NAT - Network Address Translation. In practice, it means that several devices have single IP address, that is accessible via Internet, and these clients can perform only «outgoing» connections.

There are several techniques, that deal with NAT. The industry-standard is ICE (Interactive Connectivity Establishment), but implementing ICE also requires a server.

To deal with it, we propose a new technique of NAT traversal, P2P-traversing. This technique generally relies on the same technical «tricks», but doesn't strongly rely on server. In fact, in case if just one user has «white» IP (e.g. doesn't use NAT to access Internet), the system proposed can operate without server.

To describe the P2P ping, first we describe the NAT Traversing technique called UDP hole punching. In a simple way, it is shown on fig. 1:

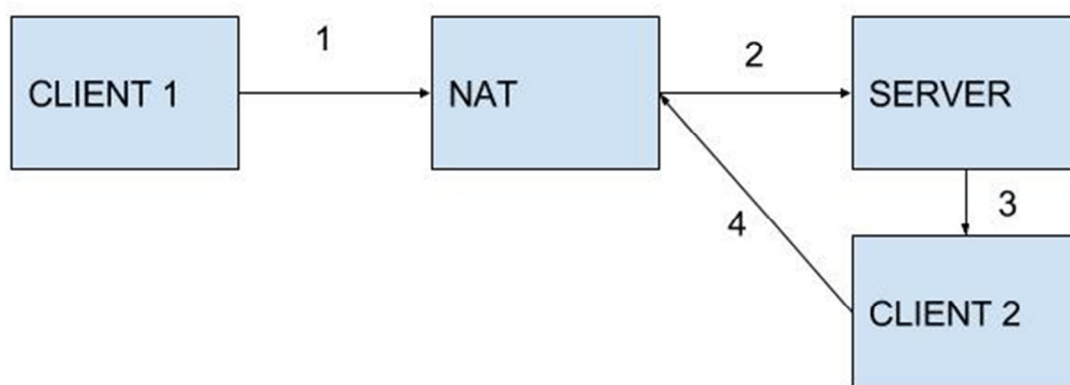


Fig. 1. General UDP Hole Punching implementation

On Fig. 1, the steps are shown with numbers 1-4:

1) Client sends UDP packet to Server. It first goes to a device that implements NAT (usually, a router)

2) Router sends this packet further to a servers, remembering, that this packet was sent be client 1 and the program port X is allocated for this request. This is done, because, if server wants to reply, it will reply to NAT device on the same port, so NAT device has to remember to which client it has to send message, came to this port.

3) The server sends the IP and port X it saw from request to Client 2 (this can be done in many ways, essential is that client 2 somehow gets this «credentials»).

4) Client 2 now sends packet to IP and port X it gathered from server. This is, actually, already a P2P communication.

This technique is called UDP hole punching and, as it seen from fig. 1, requires a server.

Now we propose the algorithm, that doesn't require server all the time, just at the very beginning. Every client implements the pinging protocol:

1) client sends UDP requests with his identifier to every IP:port he knows, in this request client puts his identifier in messaging system (e.g. his public key). This is done periodically, e.g. several times per second

2) client listens for incoming UDP requests, and, when he receives one, he makes a record in his local database, containing triplet ID:IP:Port (or updates, if such record already exists). This local database should be persistent (meaning, it should keep it's state even if application restarts).

3) periodically, client sends to all known triplets ID:IP:Port request to share these triplets

4) client listens for incoming requests to share triplets ID:IP:Port. When received, he sends his local table back

5) if client received UDP request and then stopped receiving them for certain period of time, this record is removed from triplets database;

Let's imagine we have 100 clients and 1 server, that is known by everyone. The term «server» is quite incorrect here, we shall further call it «node». It runs the same software, as clients, but it's IP address is known by clients at the very beginning.

Right after start, all the clients shall ping server and server will collect all the ID:IP:port triplets, according to steps 1 and 2 of the protocol. According to steps 3 and 4, server will share this data among everyone.

What happens if we know turn off the server and one client will change it's IP:

1) All the pings to this client will faile

2) It will be able to ping another clients that didn't change their IP

3) These client will update triplets ID:IP:port for this client

4) According to steps 3 and 4 of the original protocol, these changes will spread across the network. According to step 5 old invalid records will be removed.

So we see, that this network, once established (precisely speaking, *sufficiently established*, because server may go down before full network table will be spread among all clients), is self-sufficient.

It must be noted here, that UDP hole punching is the most reliable, but not the only technique of NAT traversing. There are 2 more techniques. One is called «router-assisted traversing» and the other - «traversing using relay net», or TURN. The first technique can be implemented locally and does not affect the described protocol. TURN, though, is not a pure «NAT traversing»: it actually uses external server as a relay. The TURN information can be added into UDP ping requests and saved along with the triplet ID:IP:port.

It is important to say, that this solution has some assumptions:

- 1) if new client comes to network, server node is needed;
- 2) if a significant amount of users would change their IP addresses, the network may not be re-established without server.

But still, the solution proposed has advantages both over traditional and P2P architectures, because mentioned requirements are rarely broken.

Next, we have to address the issue of offline messages. Actually, we need just several additions to address it. Two things needed to be implemented is routing and extending the protocol to send/receive message and store it to local DB.

The algorithm looks like:

- 1) Find the shortest route to client, except tested ones (on the first run it is direct connection), and take the closest to the destination online node
- 2) If there are no online nodes at all, go to step 1, else - send message

The protocol is extended:

- 1) Client can send message
- 2) Client listens to messages, if received, it goes to DB if message is for another user, or it is shown to user if the message is for this user
- 3) Client can push message closer to its receiver ones he sees that closer node to the receiver is available
- 4) When pushed, message is removed from client's persistent storage

According to this algorithm, message will be pushed as close to the client as possible, and once he'll be online and message will be in 1 push before him OR closer paths will be found, the message will be delivered. To ensure reliability of this solution, messages can be pushed in several routes simultaneously. Also, each client can have a reliability rating, depending on the overall uptime.

The proposed algorithm successfully resolves the issue with asynchronous offline messaging in P2P networking.

It is important to say, that this solution has some disadvantages:

- 1) it creates additional network traffic;
- 2) it is slower, than on traditional architecture.

But still, it overcomes more serious issues of traditional and P2P architectures.

The last issue to address in the proposed application is perfect forward security.

Let's see the traditional Diffie-Hellman algorithm:

- 1) Alice and Bob agree on a finite cyclic group G of order n and a generating element g in G . (This is usually done long before the rest of the protocol; g is assumed to be known by all attackers.) The group G is written multiplicatively.
- 2) Alice picks a random natural number a , where $1 \leq a < n$, and sends g^a to Bob.
- 3) Bob picks a random natural number b , which is also $1 \leq b < n$, and sends g^b to Alice.
- 4) Alice computes $(g^b)^a$, which is a secret key.
- 5) Bob computes $(g^a)^b$, which is a secret key.

The first step does not actually includes protocol - it is «hard-coded» during application development.

As it can be seen, stages 2 - 3 require communication, while we need it to be done, when one client is offline. Let's assume, that Alice is offline when Bob wants to communicate with her. Also, let's assume, that before going offline, Alice performed the following:

- 1) Generated a bunch of random numbers a [$a_1, a_2, \dots, a_m$];
- 2) Using this array, created array of half-keys [$g^{a_1}, g^{a_2}, \dots, g^{a_m}$];
- 3) Saved them locally in duplets $\text{KeyID-}i : g^{a_i}$
- 4) Sent them to nearby clients (whose IP addresses and ports are known to her).

Now, Bob have to do following:

- 1) Find online clients, who store Alice's half-keys
- 2) Get any of them, preferably - random, with it's Key-ID
- 3) Make from it a secret key (generate random number b , make $(g^a)^b$)
- 4) Send message, encrypted on this key, and include Key-ID and g^b into this message

Now Alice, when online, will be able to receive message, find her half-key using Key-ID, complete it to a full-key using g^b , that Bob attached to message.

Of course, this scheme is quite primitive, Alice's signature is required on every half-key distributed into the system, but still it allows to implement perfect forward security in asynchronous offline messaging.

So, the proposed solution successfully resolves general challenges of instant messaging: significantly reduces impact of server denial in case of accident or malicious attack, allows offline messaging in peer-to-peer communications, allows perfect forward security in chats when one user is offline.