LIBUHP (part1)

punch protocol description

2017 07 24
Status: draft #3
Author: Franck
<franck.rupin@gmail.com>
<http://www.holepuncher.io>
<#holepuncher on irc.freenode.net>
<twitter: @gurwallbzh>
<github:https://github.com/0xFranckx0>

Project source: Status **in progress**.
https://github.com/0xFranckx0/udpholepuncher/tree/protocol

Abstract

This document is about a simple protocol to punch NAT and Firewalls as it is defined and implemented by libuhp. It works only over UDP and allows realtime applications, vpn solutions or even diagnosis tools to easily implement a punch for NAT traversal. In this paper we will first remind basics about NAT and what we need to take care about for our protocol. Then we will present protocol's messages, mechanisms description and internal operations. At the end, we will present results across different network topologies and kind of NAT.

I Introduction

LIBUHP is an Udp Hole Puncher c library aimed to provide an easy way to punch NATs and firewall. Easy to use and provided with a program *punchctl*. LIBUHP could not fit with your requirements, although, for more complex needs you should take a look at ICE implementations such as PJNATH. For libuhp we focused our development over [RFC5128] and [RFC2993]. Part 2 of this paper will be dedicated to tests and results.

II NAT's basics

For exhaustive terminology about NAT you should refer to [RFC2663]. NAT is an acronym meaning Network Address Translation. Thus basic operation performed by a NAT system is to translate a network address from a realm to another one. NAT is widely used because it comes with many advantages. In the lack of IPv4 context NAT is a way to share one public IPv4 with many private IPv4 addresses. Another advantage for NAT is hiding network topology. Indeed, from outside the realm, it becomes hard to discover the network topology behind the NAT. So, is NAT perfect? There is a huge drawback by using NAT. As a host can't be reached directly from outside, if a server is hosted behind NAT it will be necessary to create a hole in NAT to be reached from a public access. Actually this issue can be solved easily. Troubles become harder to solve when you try to connect 2 clients behind NAT that want to communicate directly. For instance browsers running WebRTC application, VOIP clients, P2P… To address that, you need to implement a protocol such as ICE to connect these two clients to each other. ICE is robust and it is mandatory for other standards (i.e. RELOAD). But sometimes you need to implement something lighter than ICE, and LIBUHP is aimed to address that purpose. Next sections will detail LIBUHP.

III Protocol

This section details the protocol. We first describe the messages involved in the protocol, then the sequences of the protocol and finally internals of the protocol.
After n retries aborted the hello will be discarded.
After x seconds without response the transaction will be discarded.
A complete transaction consists in Hello/ack/bye/bye

III.1 Messages

III.1.1 Field description

Type: hello, ack, bye, cancel
Punchid: is a positive integer that identifies the sequence.
Count: is used to identify retransmissions
Epoch: relative time.

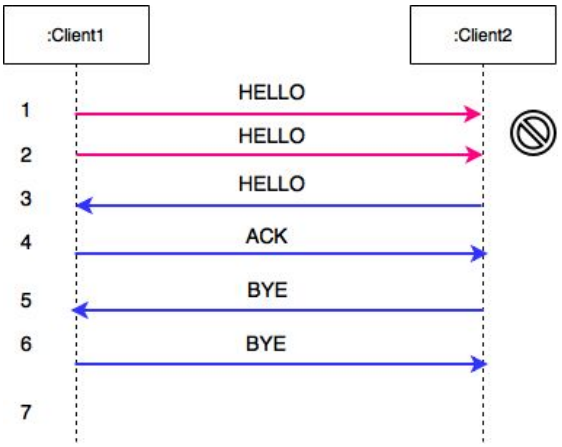
All messages are sent over UDP in JSON format.

III.1.2 JSON

```
{
        "type" : STRING,
        "punchid" : INTEGER,
        "count" : INTEGER,
        "epoch": INTEGER
}
```

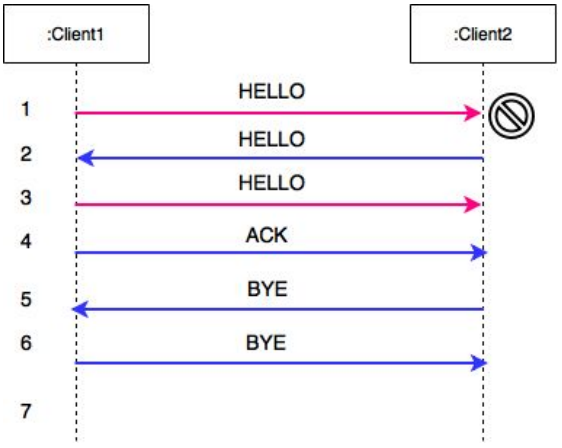
III.2 regular punch with port preservation

This is the easiest use case to implement. Here Client 1 try to reach client 2 on a predefined port that both of them have negotiated in a third part way (i.e: rendez-vous server, configuration file etc).



1- First hello failed because there is no hole in the client2 NAT. But there is one on client1 NAT.
2- Client1 continue to send hello until it can reach client2
3- When client2 send a hello, it reaches client1 because client has already opened its NAT on this port.
4- Client1 receives the hello from client2 and stops trying to send hello. Instead it replies with ACK.
5- The transaction will finish by sending BYEs.

III.3 concurrent punch with port preservation



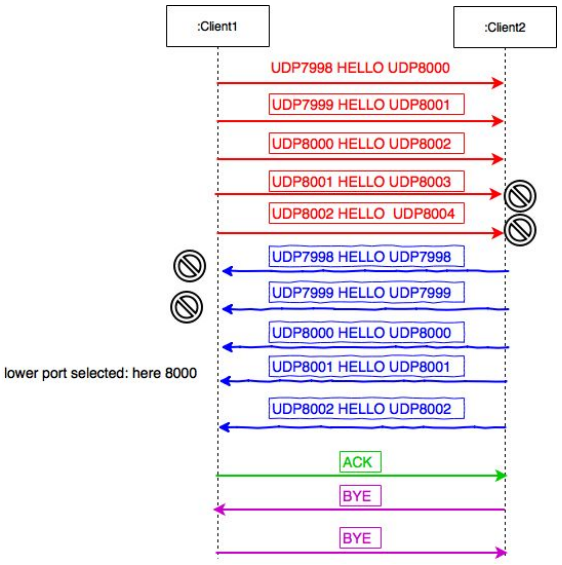
1- First hello failed because there is no hole in the client2 NAT. But there is one on client1 NAT.

2- When client2 send a hello it reaches client1.
3- It is possible that client1 just before to receive the hello from client2 sent a new hello to client2 and this one should reach client2 because its NAT is now open on this port.
4- To avoid concurrent transactions, the clients will register the HELLOs in a table, the following simple rule will be apply to discard the HELLO with the lowest punch id. Although, there is only one ack sent back. In the sequence above this is client1 that sends back a ACK.
5- The transaction will finish by sending BYEs.

III.4 Regular punch with no port preservation

In the following examples multiple HELLO messages can be sent simultaneously by client 1 et 2, and each of hello can be retransmitted multiple times. We consider that NAT can modify the port without the client have knowledge about such a modification.
In the examples "***UDP7998 HELLO UDP8000***" means that the client sent UDP HELLO packet over port 7998 and NAT modified to UDP 8000. In a hole puncher viewpoint the hole will be on UDP port 8000. NAT is responsible to translate the packet to the origin port (ie 7998).

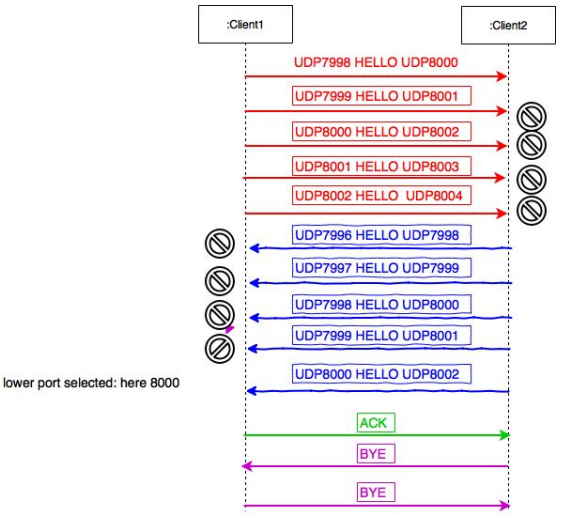
First use case:



client 1 and client 2 want to reach each other on port UDP8000
client 1 and client 2 will try on to reach on a range UDP[7998-8002]
client1 is behind a NAT that rewrites port with a rule of n + 2.
client 1 will be reachable on UDP[8000-8004]
Client 2 is behind NAT that doesn't modify port. client2 will be reachable on UDP[7998-8002]

The common external ports for communication between both will be UDP[8000-8002].
For client 1 it means it should send packets over UDP[7998-8000] while client 2 will send packets over UDP[8000-8002].
client 1 and client 2 will choose the port on which the hello with the lower sequence number was sent and drop all the others.

Second use case:



client 1 and client 2 want to reach each other on port UDP8000
client 1 and client 2 will try on to reach on a range UDP[7998-8002]
client1 is behind a NAT that rewrites port with a rule of n + 2. client 1 will be reachable on UDP[8000-8004]
client2 is behind a NAT that rewrites port with a rule of n - 2. So client 2 will be reachable on UDP[7996-8000]

The common port for communication between both will be UDP[8000] this is external port. For client 1 view point it means it should send packets over UDP[7998] while client 2 should send over UDP[8002].

III.5 concurrent punch with no punch preservation.

[To write]

III.6 Master election

The master will be the HELLO origin.

IV Implementation

IV.1 Transaction table

Each client manages a transaction table. This table is used to figure out what transaction is still valid, which port should be selected to keep open holes between NAT, to decide who is the master. The table consists into a list of transaction structure defined as follow:

```
struct transaction {
      int origin;
      int master;
      int status;
      int punchid;
      int timestamp;
      char *peer;
      int port_peer;
      int asymmetric;
      int retry;
};
```

Operations that trigger table operation:

1. Sending a HELLO:
2. Receiving a HEllO:
3. Sending a ACK:
4. Receiving a ACK
5. Sending a BYE
6. Receiving a BYE
7. A Transaction completed

Type of operations:

1. Create an entry
2. update an entry
3. delete an entry

4. search an entry
5. Read an entry

Table operations on HELLO reception:

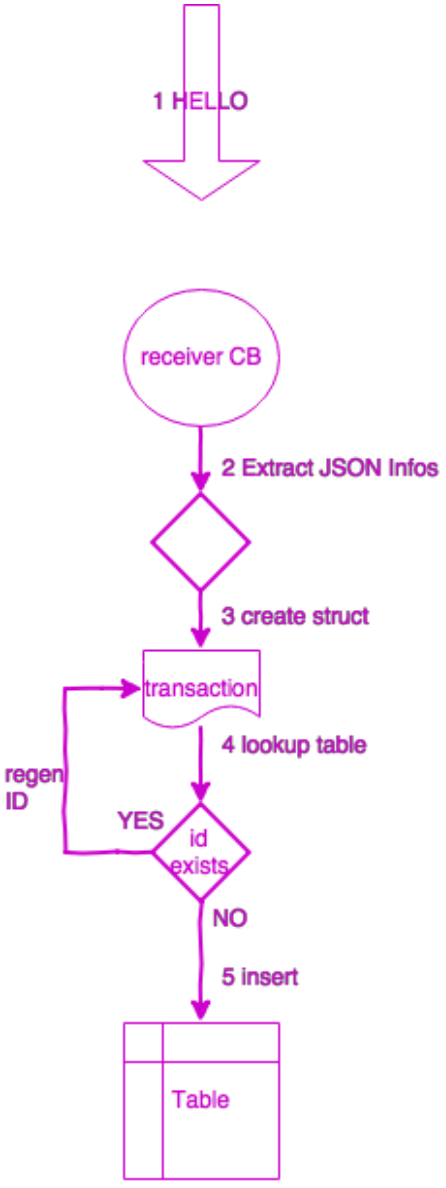


Table operations on HELLO sending:

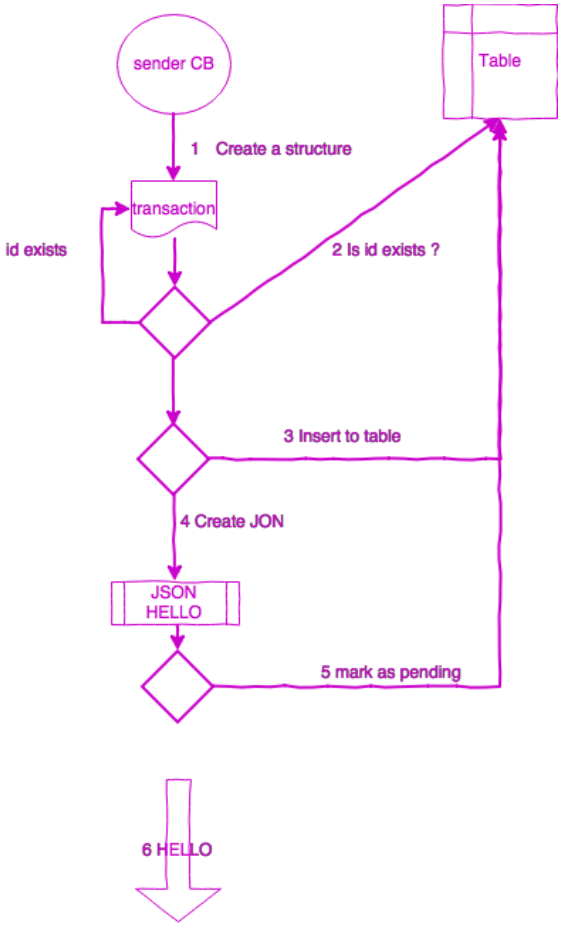


Table operations on ACK reception:

[To Write]

Table operations on ACK sending:

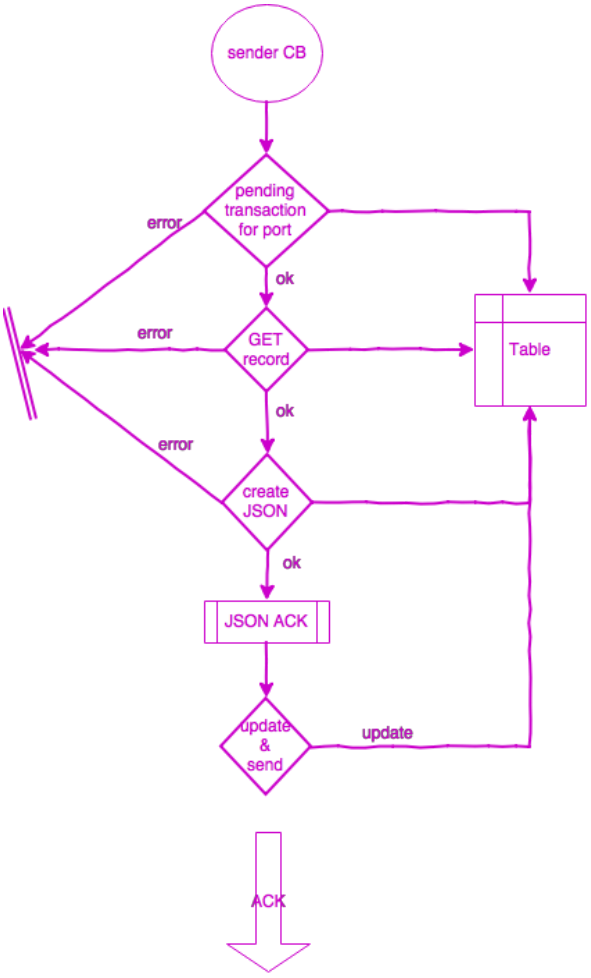


Table operations on BYE reception:

Table operations on BYE sending:

IV.2 socket selection

Once at least two transactions are completed, and there are at least two holes in NATs, the master will chose a socket associated with port to hold on future connections and keep the hole alive.
If there are two masters (one on each side), associated with two different sockets, each client will drop the older (timestamp) and keep just the newer.
If only one transaction is completed, each client drop all other transaction and select the socket associated with the hole.

IV.3 Hello selection

Hello selection is required when more than one transaction are in progress and at least one is completed. This is described in section III.3.
To HELLO selection a transaction need to be complete it means that each stakeholder has sent and received a BYE.
All uncompleted transactions must be closed and deleted.
If two transactions are completed the following rule must be apply: the client selects the transaction with the smaller transaction_id. The others completed HELLOs must be deleted!

IV.4 Transaction dropping

Once at least one transaction is completed, the table need to be cleanup by dropping all transactions.

IV.5 Dirty transaction

If a transaction became dirty, because there is incoherence between messages received and

messages expected as next step by the protocol definition, the transaction must be marked as dirty. The process should no longer take care of it and the next operation on a dirty transaction will be a drop.

IV.6 sender algo

```
table = lookup_table(port)
msg = next_message(table)
if (msg != NULL) {
update_table(table, msg)
json = msg2json(msg)
send_JSON(json)
} else {
    /* Punched */
    struct socket_punched * = get_sock();
    free(event)
    retunr ( socket_punched);
}
```

IV.7 receiver algo

```
rcvfrom(&buf)
msg = json2msg(buf)
if (msg != NULL) {
table = lookup_table(port)
update_table(table, msg)
} else {
   /* Punched */
    struct socket_punched * = get_sock();
    free(event)
    return ( socket_punched);
}
```

V DTLS support

This is an optional support. If you don't want to implement dtls, you could, just after a punch, make the channel secure by setting up dtls with libuhp. As the punch returns a socket you just need provide essentials cryptographic material to the lib as argument and then libuhp will secure the sockets with DTLS. Just remember that the libuhp context is peer2peer, so we just implement dtls for two peers that want to communicate each other, we needn't to address huge amount of concurrent requests addressed to a peer, but just a bidirectional secure channel.

References