Tunnel UDP over TCP

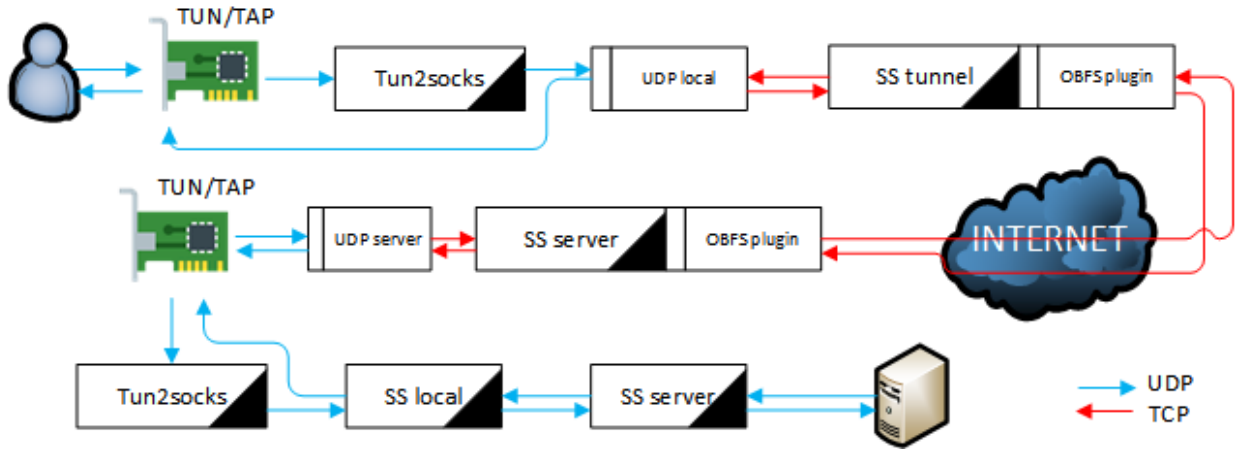
Requirements

Abstract

There are pre-existing and established TCP tunneling system using Shadowsocks(obfs plugin used) and tun2socks, and proved that this can be also used as a basis of UDP tunneling system we are going to develop now.

As we discussed, only UDP local and server will have to be newly developed for the entire system, and the rest work is to integrate all subsystems to configure the whole UDP tunneling system.

The following picture shows how UDP traffic is exchanged via the UDP tunneling system between client and target server.



UDP traffic are wrapped/unwrapped in UDP local/server, they are plugin of tun2socks or just another independent daemon.

When traffic arrives in tun2socks, it trying to connect to SOCKS server to resolve destination address, so that it can forward the traffic into gotten address.

As tun2socks already supports both TCP and UDP forwarding(normal running method for TCP, –socks5-udp option for direct UDP forwarding), we can implement UDP tunneling over TCP connection by adding UDP local and server. They will be used to encapsulate in and out the UDP traffic as shown in above picture.

Tun2socks is to only forward traffic from TUN device to SOCKS server, but not vice versa. That is, reply traffic from SOCKS sever doesn't pass through it. It goes to TUN device directly because the destination address of the traffic are already clear, it is TUN device's address.

So our udp local and server have to listen and catch, process UDP traffic between tun2socks and ss server, this is what we have to implement first. This is the core of the system, actually.

UDP local & server

They are mainly responsible for wrapping/unwrapping UDP traffic for tunneling them over TCP connection. They have to listen between tun2socks and server, to catch UDP traffic from tun2socks.

There is already developed udp-tunnel system in https://github.com/rfc1036/udptunnel.

It forwards UDP packets on TCP connection after the handshake is succeeded between client and server.

Following is simple test for its working.

nc -u 127.0.0.1 1234 --> udptunnel 127.0.0.1:1234 127.0.0.1:3456 --> udptunnel -s 127.0.0.1:3456 127.0.0.1:5678 --> nc -u -l 5678

This can be used almost directly in our system after a little modification.

So the pseudo working way can be like this;

nc -u 132.238.45.56 5001 --> tun device(10.0.0.1/24) --> tun2socks (netif-addr:10.0.0.2 netif-netmask:24 socks-server-addr: 127.0.0.1:1234) -->udp-local(binding:127.0.0.1:1234 destination:127.0.0.1:1080) -->ss-local/ss-tunnel(local_address:127.0.0.1 local_port:1080 server:96.126.127.239) --> ss-server -->udp-server(binding: 96.126.127.239:3456 destination:132.238.45.56:5001) --> nc -u -l 5001(on the target server 132.238.45.56)

132.238.45.56 and 5001 are IP address and port of target server to where the client is going to connect

The following lines show how the packet's protocol and source/destination ip address is changed while it is being transmitting.

No	Protocol	Source IP:Port	Destination:port	Status	Packet Location
1	UDP	10.0.0.1:56434 (port is random)	132.238.45.56:5001	Initial	client
2	UDP	10.0.0.1:56434	127.0.0.1:1234	While SOCKS request is sending to udp-local	tun2socks
3	TCP	10.0.0.1:56434	127.0.0.1:1080	Packet is wrapped by TCP and SOCKS request is resolved for udp-server's binding address	udp-local
4	TCP	10.0.0.1:56434	96.126.127.239:3456	SOCKS request was for the binding address of udp-server	ss-server
5	UDP	10.0.0.1:56434	132.238.45.56:5001	Packet is unwrapped and forwarded to target server	udp-server

As shown in the above diagram, when packets arrives in tun2socks, tun2socks sends SOCKS request to the address specified by --socks-server-addr option to resolve the the arrived packet's destination address. At the moment, --socks-server-addr is the address of udptunnel client's, not ss-local/ss-tunnel's.

So udp-local relays these SOCKS requests to real local SOCKS server. But the packet's destination address is changed to udp-server's binding address to let the packet go to udp-server automatically after they are passed via ss-server. Of course, UDP packets are wrapped to TCP.

Then udp-server will change the destination address into real target server's one, so that it can go to target server after unwrapped into UDP. Wait, how does udp-server know the target address? udp-local will send it via specialized TCP connection.

So our udp-local/server will have 3 main sessions for;

1. exchanging wrapped UDP packets

2. sending packets' real destination address from udp-local to udp-server

3. exchanging heartbeats to each other for session management

Routing TCP and UDP traffic into different path

Tun2socks is based on a single-threaded event-driven architecture, it reads on event loop for checking and executing new events. All events(causes operations finally) occur asynchronously, it is difficult to describe its working way as a clear flowchart.

To get a clear point of our tun2socks customizing, we need to check the internal architecture of tun2socks at the present.

In case of TCP forwarding, when TCP connections arrives in netif interface(internal virtual network interface created by Lwip) which was listening on TCP connection accept them and create respective TCP client objects according to each connection. After this point, these TCP clients will create their own SOCKS Client object to connect to the SOCKS server. Then, once the packets arrive in TUN device, they will be forwarded into netif and finally be sent to SOCKS server after SOCKSified.

In case of UDP forwarding, UDP traffic are directly processed when they arrive in TUN device, SOCKS Client and UDPSocket object are created as soon as the traffic is checked for validation. And they are finally transmitted over UDP connection directly.

The original tun2socks are designed to forward traffic into only one SOCKS server. But we have to update this so that it can forward TCP and UDP traffic into different servers.

Extra subsystems

The rest elements are already ready as in TCP tunneling system, once UDP local and server is finished, they all will be integrated into the entire system and tested.

Requirement highlights

1. Please generalize the design of udp-local/server so that they can be used to work with other network library in addition to Shadowsocks.
2. tun2socks customization for client side.

   Related to the above item 1, a little bit customized tun2socks can invoke udp-local so that the clients can totally transparently forward their both UDP and TCP traffic to tun2socks (i.e. tun2socks --> udp-local), otherwise clients need discriminate UDP/TCP to let them go to different route, however, the proposed TCP/UDP data flow (i.e. udp-local --> tun2socks) should be also supported, which makes our usage flexible.

   Priority wise, tun2socks --> udp-local in client side is preferred firstly, udp-local --> tun2socks can come afterwards. In server side, udp-server --> tun2socks is fine, there's no need to support tun2socks --> udp-server. Since we can easily create and mount a tun devices on Linux and Windows, moving udp-local after tun2socks will result in easier integration. Otherwise, we need write another interception layer to let clients route TCP and UDP differently, which is complicated. Moreover, on higher version of macOS and iOS, system extensions is not allowed to be called, NEPacketTunnelProvider has been implemented to route all traffic to tun2socks, so tun2socks --> udp-local will deliver the uniformed UDP data flow for all platforms including Android. As a result, like other ordinary VPNs, customers will be faced with the TUN devices when they configure settings, which is done automatically by client softwares. Note that NEPacketTunnelProvider, tun device management, and VPN settings are out of scope of this work. They are implemented by the VPN client.

   Note that the revision (i.e. tun2socks --> udp-local) needs a bit customization of tun2socks to route UDP and TCP traffic to different path.

3. tun2socks customization for server side.

   In the meantime, udp-server ----named pipe or domain socket----> tun2socks will possibly need udp-server call tun2socks library instead of daemon process since traffic can't be directly sent to tun2socks daemon that has no listening port for incoming traffic except virtual tun device and router to which only IPs are assigned and attached.

   This requires that tun2socks is customized a bit for the server side, i.e. between udp-server and tun2socks is a pipe where data is written and read by udp-server and tun2socks, respective, and vice versa. Specifically, tun2socks can be called and launched and set up from udp-server on demand.

4. Pair launched udp-local and server is not mandatory. They can be run as daemon and library as well separately and independently. So the session management is quite important in this case. In other words, tunneling UDP over TCP should still work even without pair launching

udp-local/server, one end is down, should be fine, relaunching it should just establish a new session to go.

5. iOS doesn't allow main process invoke a sub-process (i.e. if udp-local can only run as a process, this would be a problem), so udp-local / server should be able to run as a library and process as well for easier integration. Priority wise, library first, then we can wrap a process if needed.
6. The whole design and implementation must be cross platformed, at least to support windows/linux/macos/android/ios. So it would be the best to implement them using C/C++. The interface exposed should be C based for easier integration into other languages.
7. the udp-local / server lib should support an async interface using callback paradigm, please refer shadowsocks.h
8. There must be some relatively good unit tests to cover the implementation. C-based unit test framework is prefered.

   Once our tun2socks are finished, requirement 1, 2, 3, 4 will be finished automatically.

Related works

UDP over TCP

udptun_nel

gnb_udp_over_tcp

TCP tunneling

These are the work of TCP tunneling we are based on to implement UDP over TCP.

tun2socks

ss-local/ss-tunnel/ss-server

obfs-local/obfs-server

Implement Consideration (UDP local and server)

Summary

Between UDP local and server, there are another server/client communication subsystem-shadowsocks.

Shadowsocks requires IP raw packets from TUN device so that it can append SOCKS5 header for SOCKS communication. So both UDP local and server have to create raw TCP packets generated from UDP traffic. These features are not implemented in above libraries. They need to be developed.

In case of TCP traffic, they will be just forwarded without any modification.

Also, they have to not only forward traffic, but also synchronize their operations with each other.

So, session management module mentioned just above will be implemented.

All tests will be done first between UDP local and server where tun2socks is enabled with respective TUN device.

After UDP system is finished, then it will be integrated into pre-configured TCP tunneling system while it is little modified (ss tunnel will be used instead of ss local) and tested.

License

Only consider commercial friendly license when libraries are chosen. The implementation in this work is not meant be open sourced immediately, but will be some time.

Libraries

Note: Using and / or import libraries as git submodules only, don't import sources.

lwIP: open-source TCP/IP stack designed for embedded systems

Libev: a high performance full-featured event loop written in C

Badvpn-tun2socks: tun2socks will be only used for performance test

Libcork: collection of useful utilities for embedded system

Some other libraries used by Shadowsocks-libev that could be useful per use are

libmbedtls

libsodium

libpcre3

libc-ares

libbloom

lipipset

Milestones

----------initial milestone proposal disagreed----------

Week 1: primary version of UDP local and server for only supports forwarding UDP traffic into TUN device (libev, libcork will be considered to use)

Week 2: implement wrapping UDP traffic into TCP using lwIP

Week 3: implement session management system, integration test of entire udp local & server

Week 4: performance test with shadowsocks system, library release

----------new milestone proposal agreed, finished within 4 weeks or so----------

1. m1: implement conversion of SOCKS, UDP and TCP, vice versa, using lwIP, the conversion serves as the preliminary and reusable components for m2. Moreover, tun2socks in client should be customized to route TCP and UDP to different path. In order to test, two ss-locals that supports TCP (i.e. without -u) and UDP (i.e. -U) respectively can be launched such as tun2socks → TCP → ss-local1 and tun2socks → UDP → ss-local2

   Unit test should be added to cover the above conversion and vice versa.

2. m2: Implementing basic udp-local/server and tun2socks customization.

   Unit test should be added to simulate and test the below part of data flow and/or the whole flow.

   nc-local → tun2socks ----> udp-local → ss-tunnel → obfs-local ……internet……
   obfs-server → ss-server→ udp-server --(pipe)--> tun2socks → ss-local → ss-server → nc-server

   primary version of UDP local and server to only support forwarding UDP traffic using customized tun2socks and shadowsocks stack (libev, libcork will be considered to use).

3. m3: implement session management system, integration test of entire udp local & server.

   Unit test should be added to simulate and test the deployment of the above components of tunneling UDP over TCP.

4. m4: performance test with shadowsocks system, library release

   Benchmark unit test should be added to for performance milestone. More unit tests may be needed to expand coverage of code from m2 and m3.

----------new milestone proposal----------

1. m1: implement primary version of udp local and server. They will catch UDP traffic from tun2socks to Socks server after SOCKS request is resolved, and then wrap/unwrap into TCP traffic.

Unit test should be added to cover the above conversion and vice versa.

2. m2: implement session management system of udp local and server, finalize the development

   Unit test should be added to simulate and test the below part of data flow and/or the whole flow.

   nc-local ⟶ tun2socks ----> udp-local ⟶ ss-tunnel ⟶ obfs-local ……internet…… obfs-server ⟶ ss-server⟶ udp-server --(pipe)--> tun2socks ⟶ ss-local ⟶ ss-server ⟶ nc-server

3. m3: tun2socks customization for routing different paths

   tun2socks in client should be customized to route TCP and UDP to different path. In order to test, two ss-locals that supports TCP (i.e. without -u) and UDP (i.e. -U) respectively can be launched such as tun2socks ⟶ TCP ⟶ ss-local1 and tun2socks ⟶ UDP ⟶ ss-local2

   Unit test should be added to simulate and test the deployment of the above components of tunneling UDP over TCP.

4. m4: performance test with shadowsocks system, library release

   Benchmark unit test should be added to for performance milestone. More unit tests may be needed to expand coverage of code from m2 and m3.

Appendix

Original Proposal

Abstract

This proposal describes how to support the UDP protocol.

Design

The core idea is to reuse all existing implementations that support the TCP protocol, but wrap UDP data packet into TCP in the client side, and unwrap TCP packet into UDP one in the server side.

Data Flow

TCP Data Flow

tcp ---> tun2socks --> ss-local --> obfs-local --> .....internet HTTPS(TCP)..... obfs-server ---> ss-server ---> target server

UDP Data Flow

udp ---> tun2socks(API) → udp-local --TCP --> ss-tunnel --> obfs-local --> .....internet HTTPS(TCP)..... obfs-server ---> ss-server ---> udp-server --UDP (through named pipe or domain socket)--> tun2socks(API) --> ss-local -- ss-server

The two ss-server in UDP Data Flow can be the same process.

Unlike ss-local, ss-tunnel can specify which server (e.g. udp-server in this case) will receive the packets. ss-local just lets ss-server to access the final target server.

Note that:

[1] tun2socks is a set of API to wrap raw TCP/UDP packet into SOCKS one.

[2] udptunnel can be used as udp-local and udp-server given the different arguments.

[3] is how to wrap raw packets into SOCKS ones or vice versa.

Special Notes

udptunnel:

udp-local and udp-server must be launched in a pair, if udp-server is rebooted, so needs udp-local.

badvpn and tun2socks:

Some dependent libraries may be required to wrap UDP/TCP packets into SOCKS ones or vice versa. Badvpn needs to be built as a whole.

References

[1] tun2socks

[2] udptunnel

[3] wrap packets into SOCKS or vice versa

Current tun2socks based TCP flow

$ sudo ./ProxyController /var/run/proxy_controller_domain_socket [-d]

This creates a domain socket to which we send command to

./ProxyController itself will create TUN device and config routing rule, e.g. create ip route that take all traffic of the computer.

This {"action":"configureRouting","parameters":{"proxyIp":"remote-ss-server-ip","routerIp":"10.0.85.1"}}

wil be sent to proxy_controller_domain_socket to configure TUN device, such as as a result

$ default via 10.0.85.1 TUN0 enp0s5 proto static metric 100

Then these will be launched,

$ badvpn-tun2socks --tundev tun0 --netif-ipaddr 10.0.85.2 --netif-netmask 255.255.255.0 --socks-server-addr localhost:1080

$ ss-local -s $SOCKS_SERVER_IP -p $SOCKS_PORT -b 0.0.0.0 -l 1080 -k $SOCKS_PASSWORD -m chacha20-ietf-poly1305 -u

You see that our current implementation for TCP and UDP is:

tun0(10.0.85.1) --> tun2socks(10.0.85.2) --> ss-local --> obfs-local

The UDP is swallowed if obfs-local is used since the latter doesn't support UDP.