# Pluggable Transport UDP Support Proposal



## Scope of Work

UDP Support for Pluggable Transports could refer to two basic concepts: allowing use of UDP for transports, and allowing use of UDP for applications. There are 4 possible configurations:

| Application | Transport | Status |
|---|---|---|
| TCP | TCP | Currently Supported |
| **UDP** | **TCP** | **Discussed in this proposal** |
| TCP | UDP | Untried, but possible |
| UDP | UDP | Possible Future Work |

All transports that are currently implemented use TCP. Therefore, this proposal will focus on adding UDP application support using the existing TCP transports. This means that the application client will send UDP packets to the obfuscation client, TCP packets will be sent between the obfuscation and the obfuscation server, and then the obfuscation server will send UDP packets to the application server.

The goal of this proposal is to add UDP support to the PT implementation. Therefore, "RFC 6062 - TCP relaying TURN extension", is not in scope. This extension, if implemented, would allow TURN to be used instead of SOCKS for proxying TCP application traffic.

There is a known issue with the performance of UDP over TCP head of line blocking. As this proposal is just a first step at enabling UDP application traffic over Pluggable Transports, potential performance issues of the implementation are outside of the scope of this proposal.

# Obfuscating Tunneling Relay

The obfuscation client and server together form an obfuscating tunneling relay that appears to the application client and application server as a proxy. Unlike a normal proxy, it must be split into two components. This is because, in the use case in which a Pluggable Transport is used, application traffic cannot transit the network between the application client and the application server due to filtering. Therefore, a traditional single hop relay will not generally work as either one side or the other will encounter filtering. With an obfuscating tunneling relay, the proxy is broken into two pieces. The transport client talks to the application client locally. The transport server talks to the application server over the unfiltered Internet. The transport client talks to the transport server using an obfuscated protocol. The application protocol is therefore tunnelled inside the transport protocol.

# Communicating with the Relay

Currently, communicating with the relay uses SOCKS5. This is sufficient for tunneling TCP applications over TCP transports. However, the goal of this proposal is to tunnel UDP applications over TCP transports. There is a version of UDP tunneling available in the SOCKS5 specification. However, it is not implemented in application clients. The alternative proposed here is to use TURN as the protocol to communicate with the relay. This is an alternative protocol to SOCKS5. It has mechanisms for proxying UDP traffic. The basic mechanism of TURN is that the application client sends UDP control packet to the TURN server to allocate bindings, which are externally routable addresses. Once a binding has been established, the application client can send UDP data packets and these will get forwarded to the destination.

TURN has a number of related specifications. The following guide shows which RFCs will be implemented as part of this proposal, as well as which will be held for possible future investigation, which are apparently not relevant to our use case, and which are officially declared to be obsolete:

- Implementing as part of this proposal
    - RFC 5766 - base TURN specs
    - RFC 5389 - base "new" STUN specs
- Not implementing as part of this proposal
    - Possible future investigation
        - RFC 6156 - IPv6 extension for TURN
        - TURN Bandwidth draft specs (http://tools.ietf.org/html/draft-thomson-tram-turn-bandwidth-01)
    - Not apparently relevant
        - RFC 5769 - test vectors for STUN protocol testing
        - RFC 6062 - TCP relaying TURN extension
        - RFC 6156 - IPv6 extension for TURN
        - RFC 7443 - ALPN support for STUN & TURN

        - RFC 6062 - TCP relaying TURN extension
        - RFC 7635 - oAuth third-party TURN/STUN authorization
        - Origin field in TURN (Multi-tenant TURN Server) (https://tools.ietf.org/html/draft-ietf-tram-stun-origin-06)
    - Officially obsolete
        - RFC 3489 - "classic" STUN

The two RFCs being implemented are RFC 5766 (base TURN) and RFC 5389 (base STUN). The RFCs on IPv6 and bandwidth are marked for future possible investigation. The other RFCs are deemed not relevant or obsolete.

# Integration with TCP Transports

Once a TURN server has been implemented, the next step is to hook it up to the existing transports. There are two parts to this: mapping the information contained in the TURN headers to the information contained in the SOCKS headers, and mapping the UDP packets to a TCP datastream.

## Relay Configuration Headers

The relay needs 3 pieces of information:

1. The obfuscation server destination address
2. Parameters for the obfuscating transport
3. The application server destination address

In the current SOCKS implementation, these are provided by parsing the SOCKS protocol headers that are placed at the beginning of the TCP stream before the data. Of these 3 pieces of information, the SOCKS protocol only has a field for the application server destination address. Therefore, the obfuscation server destination address and parameters for the obfuscating transport are encoded by overloading the username and password fields in the SOCKS header normally used for authentication. These are in a format specific to the obfuscation client and must be encoded and placed there by the application client. In theory, they could be pre-encoded and given to the user to place in the username and authentication fields when configuring the application to use the SOCKS proxy. In practice, the application client is aware that it is using a non-standard SOCKS proxy and encodes the information and places it into the username and passwords fields. The obfuscation client parses these fields and extracts the information. It is then passed to the transport using the transport API.

In order to enable UDP application support for existing transports, this same information that is encoded in the SOCKS headers information must instead be encoded in the TURN protocol. Fortunately, TURN is similar to SOCKS in that there are also username and password fields normally used for authentication. The necessary information can therefore be encoded in a similar way into these fields. Also like SOCKS, the TURN protocol has a dedicated field for the application server destination address. The TURN implementation can therefore behave in a similar manner to the SOCKS implementation, parsing the username and password fields,

extracting the necessary information, and passing it to the transport API. One significant difference between the way these fields are handled in SOCKS and TURN is that in SOCKS the headers are added to the beginning of each TCP stream. In TURN, they are added to each UDP packet.

## Relaying Data

In the current implementation, once the SOCKS headers have been parsed, the SOCKS server no longer handles the connection. The TCP connection is passed to the transport and the transport is tasked with handling it. The transport therefore does not have direct access to the SOCKS headers, as it receives the stream after it has been advanced past the end of the headers.

The situation with TURN is somewhat different. There is no stream, only individual UDP packets. Each packet could be either a control or data packet, and each packet could also be bound for a different destination. Therefore, the information which for TCP streams would be included in the SOCKS header, in the case of TURN is included as a header on each UDP packet.

When receiving a UDP packet that is a data packet, the TURN server strips the headers, retaining the data, and then sends the packet through the transport. However, since the existing transports use TCP, the UDP packets need to be packaged into a stream. Fortunately, RFC 5389 already specifies a framing protocol for transmitting UDP packets over TCP transport streams, specifically Section 7.2.2.

# Configuring the TURN Server

The current PT implementation provides a SOCKS server. In order to add UDP support, a TURN server will be added. However, there is currently no mechanism for supporting both SOCKS and TURN at the same time. Specifically, for an incoming connection to a obfuscating server there is nothing in the communication protocol between the obfuscating client and obfuscating server to indicate whether the obfuscating server should connect to the application server using TCP or UDP. Therefore, for this initial work, a command line flag will be added to the PT implementation that specifies TURN mode (SOCKS mode is the default). The obfuscating client will run either a TURN or SOCKS server, depending on the flag. The obfuscating server will makes either UDP or TCP connections to the application server, depending on the same flag.

# TURN Relay Exit

The original PT specification was designed for the use case where Tor is being used as both a client and server. On the client side, Tor speaks the custom SOCKS protocol to the PT client, including necessary configuration information in the user and password fields. On the server side, the PT server proxies incoming connection streams to a local Tor server. The destination server address is therefore static and global. It is configured at runtime when the PT server is launched. Connection to the application server is handled independently of the PT server, through negotiation between the Tor client and Tor server using the Tor protocol. This negotiation is opaque to the PT client and server.

In the TURN implementation, there is no local Tor server. UDP supports is intended for different use cases other than Tor. Additionally, Tor cannot receive or send UDP traffic, so it cannot be used on either the client or server. On the client side, the role of Tor can be replaced with any TURN client that can provide the necessary information in the user and password fields. On the server side, something needs to take the role of Tor in making a connection to the destination application server. The proposed solution is to provide normal TURN proxy capabilities as part of the PT server. Like any TURN server, the PT server receives UDP packets and forwards them to the destination application server.

# Obfuscating Client to Obfuscating Server Communication

In the current SOCKS implementation, there is no explicit communication protocol between the obfuscating client and obfuscating server. Each transport will use a transport-specific method of communication. However, there is no standard mechanism to pass the destination address of the application server to the obfuscating server. In the current use case for the SOCKS server, the obfuscating server acts as a transparent proxy to a local Tor server. The destination server address is therefore static and global. It is configured at runtime when the PT server is launched.

In the TURN implementation, each UDP packet contains headers specifying the application server destination address, the PT server destination address, and transport-specific parameters. The PT client receives these packets and strips the PT server destination address and transport-specific parameters, leaving the data and the header specifying the application server destination address. These stripped packets are packaged into a datastream and delivered to the PT server over the chosen obfuscated transport protocol using TCP. On the PT server, the obfuscated transport connection using TCP is received and the packets are unpacked from the datastream. They are then parsed as normal TURN packets. Since both control and data packets are relayed, the TURN server that is part of the PT server can parse

the packets and relay them just as any TURN server would, using the PT client as an application client. The replies are similarly packaged and sent over the transport and then relayed from the PT client to the application client.