Pluggable Transport Specification

Version 2.0, Draft 1

Abstract
Pluggable Transports (PTs) are a generic mechanism for the rapid development and deployment of censorship circumvention, based around the idea of modular transports that transform traffic to defeat censors.

There are two ways to use transports. A set of language-specific APIs are defined to use transports from within an application, running the transports in the same process as the application. Transports can also be run in a separate process. In this case, a transport provider sub-process is created and a custom inter-process communication (IPC) protocol is used to communicate between the application and the transport provider sub-process. This document specifies APIs for Go and Python, as well as the sub-process startup, shutdown, and IPC protocol.

Table of Contents

1. Introduction

This specification describes a way to decouple protocol-level obfuscation from an application's client and server code, in a manner that promotes rapid development and reuse of obfuscation techniques for use in circumvention tools.

Pluggable Transports can be integrated into an application in two different ways. The first way is by using a language-specific API for accessing transports. This document specifies APIs for Go, Javascript, and Python. Similar APIs could be designed and implemented for other programming languages. The goal of the API is to provide a socket-like object that can be used by the application for communication over a transport. The API approach offers minimal complexity to application developers that are using languages for which an API implementation exists. The second way to employ transports in an application is by utilizing helper sub-processes that implement the necessary services that handle the censorship circumvention. This document specifies how to launch and communicate with these sub-processes.

1.1. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Architecture Overview

```
+------------+                     +---------------------------+
| Client App +-- Socket-like API --+ PT Client Library         +--+
+------------+                     +---------------------------+  |
                                                                  |
            Public Internet (Obfuscated/Transformed traffic) ==>  |
                                                                  |
+------------+                     +---------------------------+  |
| Server App +-- Socket-like API --+ PT Server Library         +--+
+------------+                     +---------------------------+
```

Figure 1. API Architecture Overview

When using the API, the PT Client is integrated directly into the Client App and the PT Server is integrated directly into the Server App. The Client App and Server App communicate through socket-like APIs, with all communication between then going through the PT library, which only sends transformed traffic over the public Internet.

```
+------------+                 +---------------------------+
| Client App +-- Local Loopback --+ PT Client              +--+
+------------+                 +---------------------------+  |
                                                              |
          Public Internet (Obfuscated/Transformed traffic) ==> |
                                                              |
+------------+                 +---------------------------+  |
| Server App +-- Local Loopback --+ PT Server              +--+
+------------+                 +---------------------------+
```

Figure 2. IPC Architecture Overview

When using the transports in a sub-process, IPC is used to communicate with the transports. On the client's host, the PT Client software exposes a local proxy to the client application, and obfuscates or otherwise transforms traffic before forwarding it to the PT Server's host. The PT Client can be configured to provide different proxy types. The PT 1.0 and PT 2.0 (Section 3.3) protocols are based on extensions of the SOCKS proxy protocol [RFC1928]. These proxy protocols allow for per-connection configuration arguments to be passed to the transports. A transparent TCP proxy mode can also be specified, which does not allow for per-connection configuration arguments. UDP modes can also be specified, including both a transparent UDP proxy mode and a STUN-aware mode. The proxy modes are specified using command line arguments when launching the sub-process.

On the server's host, the PT Server software exposes a reverse proxy that accepts connections from PT Clients, and handles reversing the obfuscation applied to traffic, before forwarding it to the Server App. The Server App will always be listening on a local loopback socket. However, there are several different options for how the Server App will behave. First of all, it could be listening for either TCP or UDP traffic. This is specified using a command line flag when launching the PT server sub-process. By default, communication will be a transparent proxy, which forwards traffic that is received at the PT server directly to the Server App. There is also an optional lightweight protocol to facilitate communicating connection meta-data that would otherwise be lost such as the source IP address and port [EXTORPORT]. Additionally, layering of proxies is possible. The Server App may itself be a proxy server and expect the forwarded traffic it receives to conform to a proxy communication protocol, for instance SOCKS or TURN.

All PT instances using the IPC method are configured by the respective parent process via a set of standardized environment variables and command line flags (3.2) that are set at launch time, and report status information back to the parent via writing output in a standardized format to stdout and error messages to stderr (3.3).

Each invocation of a PT MUST be either a client OR a server.

PT client forward proxies MAY support any of the following proxy modes: PT 1.0 with SOCKS4, PT 1.0 with SOCKS5, PT 2.0, transparent TCP, transparent UDP, or STUN-aware UDP. All PT client forward proxies MUST support at least PT 1.0 with SOCKS 5 to provide backwards compatibility with older Client Apps. Clients SHOULD prefer PT 2.0 over PT 1.0 and PT 1.0 with SOCKS5 over PT 1.0 with SOCKS4.

3. Specification

3.1. Pluggable Transport Naming

Pluggable Transport names serve as unique identifiers, and every PT MUST have a unique name.

PT names MUST be valid C identifiers. PT names MUST begin with a letter or underscore, and the remaining characters MUST be ASCII letters, numbers or underscores. No length limit is imposted.

PT names MUST satisfy the regular expression "[a-zA-Z_][a-zA-Z0-9_]*".

3.2. API Interface

3.2.1. Goals for interface design

The goal for the interface design is to achieve the following properties:

- Transport implementers have to do the minimum amount of work in addition to implementing the core transform logic.
- Transport users have to do the minimum amount of work to add PT support to code that uses standard networking primitives from the language or platform.
- Transports require an explicit destination address to be specified. However, this can be either an explicit PT server destination with the Server App is already known implicity (the case with obfs4), or an explicit Server App destination with the PT server destination already known implicity (the case with meek).
- Transports may or may not generate, send, receive, store, and/or update persistent or ephemeral state.
    - Transports that do not need persistence or negotiation can interact with the application through the simplest possible interface
    - Transports that do need persistence or negotiation can rely on the application to provide it through the specified interface, so the transport does not need to implement persistence or negotiation internally.
- Applications should be able to use a PT Client implementation to establish several independent transport connections with different parameters, with a minimum of complexity and latency.
- The interface in each language should be idiomatic and performant, including reproducing blocking behavior and interaction with nonblocking IO subsystems when possible.

3.2.2. Abstract Interfaces

These are high-level/pseudocode descriptions of the interfaces exposed by different types of transport components. Implementations for different languages and platforms may be radically different, so long as the functionality is ultimately equivalent. In some languages, the natural way to *provide* this functionality may be different from the natural way to *consume* it, so two language-specific interfaces may be required.

By convention, we use the term *opaque* to indicate a key-value map with string keys whose names are specific to each transport. Some common key names (e.g. “host”, “port”) may form a convention used by various transports where they make sense.

3.2.2.1. TCP Client

The basic TCP client interface consists of

- **Transport Factory**, which takes a read-only opaque argument (the **client configuration**) and returns a set of **Connection Factories**
- **Connection Factory**, which takes a read-only opaque argument (the **connection settings**) and produces a working connection similar to the environment’s native TCP socket type
    - Labeled with a **connection factory type name** to distinguish it from other factories in the set.
    - Depending on the transport, the ConnectionFactory may also requires an argument indicating the TCP **destination endpoint**.
    - The connection object is extended to have an additional **get stats** method

Some transports may also have additional mandatory or optional arguments to the **Transport Factory**.

- **local**: long-term local storage (e.g. a filesystem path or database object)
- **signals**: a pipe for sending messages to the server, and receiving messages from the server, out of band. Transmission is unreliable, and may require significant manual action for each message (e.g. copy and paste into an e-mail).

3.2.2.2. TCP Server

Similar to the TCP client, there is a basic interface for stateless, non-negotiating clients, and a more complex interface for transports with more complex needs. The basic interface is

- A **Server Factory**, with methods
    - **generate configuration**, which returns a pair of opaque objects: the **client configuration** and the **server configuration**
    - EITHER **listen**, which takes a **server configuration** and returns an object resembling a native listening server socket (plus **Server Manager** as a mixin)
    - OR **proxy**, which takes a **server configuration** and returns a **Server Manager**
- A **Server Manager**, which represents a live, listening server. Its methods include

    - **stop server** (self-explanatory)
    - **get stats**, which returns connection statistics in a standard format

Some transports may require additional arguments to the **listen** or **proxy** methods:

- **local**: long-term local storage
- **signals**: a message-based pipe for sending messages to the client, and receiving messages from the client, out of band. Transmission is unreliable, and may require significant manual action for each message (e.g. copy and paste into an e-mail).

3.2.2.3. UDP Peer

Same as a TCP Client with no connection settings (UDP is connectionless), but replace "server" with "other peer".

3.2.3. Example SSH Key-based Transport

Consider a TCP transport for clients that uses SSH as the transport. This transport allows connections to all hosts, and so requires the Server App endpoint as part of the configuration. Clients must authenticate to the server using their locally-generated public key. In this example, we consider this transport as part of a graphical desktop application.

3.2.3.1. Client side

- The **client configuration** consists of the server's host, port, and the public key fingerprint. It might be distributed as JSON in an e-mail.
- The **Transport Factory**
    - Checks if a key named "sshKey" is present in **local**.
        - If so, retrieve the "sshKey" value
        - Otherwise,
            - generate an SSH keypair
            - store it as "sshKey" in **local**
            - emit a message on **signals** providing the client's key fingerprint
                - This causes the application to surface a notification requiring the user to pass a provided string to the server operator.
    - Spawns an SSH client process with "ssh -D 54321" to create a SOCKS proxy on localhost
        - If the connection fails with an authentication error, retry every 10 seconds, on the assumption that the fingerprint hasn't yet been processed by the server operator
- The **connection settings** are empty
- The **Connection Factory** opens a socket through the localhost proxy on port 54321 to the specified destination endpoint

3.2.3.2. Server side

The server side of this transport could be performed entirely manually by a sysadmin. However, a lightly scripted version that runs its own instance of sshd could work as follows

- The **Server Factory** starts a copy of sshd as an unprivileged user, bound to an external port but not accepting any connections
    - **get configuration** returns a JSON blob containing the system's public IP address, the port on which sshd is listening, and sshd's public key fingerprint
    - **proxy** configures sshd to allow proxy connections (still no shell access).
- The **signals** pipe receives clients' fingerprints and appends them to sshd's allowed-keys configuration file.

3.2.4. Go Interface

The Pluggable Transport Go API provides a way to use Pluggable Transports directly from Go code. It is an alternative to using the IPC interface. The Go API may be an appropriate choice when the application and the required transport are both written in Go. When either the application or the transport are written in a different language, the IPC interface provides a language-neutral method for configuring and using transports running in a separate process.

This Go API is a proposal. It has not yet been implemented and the current Go implementation of Pluggable Transports, obfs4proxy, does not follow this API. This draft is primarily intended for discussion about the best choices for an idiomatic API for the Go language.

This API specification is divided into three parts. The "Modules" section provides documentation of the types and methods provided by the Pluggable Transports Go API. The "Implementing a Transport" and "Using a Transport" sections then provide context on how the API is used.

3.2.4.1. Modules

The Pluggable Transports Go API provides one module: pt. It is intended to be used as a replacement for the net module provided by the Go standard library. The API mirrors that of the net library.

3.2.4.1.1. Module pt

```
// The Transport interface provides a way to make outgoing transport connections and to accept
// incoming transport connections.
// It also exposes access to an underlying network connection Dialer.
// The Dialer can be modified to change how the network connections are made.
interface Transport {
  // Dialer for the underlying network connection
  networkDialer() *Dialer
  // Create outgoing transport connection
  (transport *Transport) Dial(address string) pt.TransportConn

  // Create listener for incoming transport connection
  (transport *Transport) Listen(address string) pt.TransportListener
}

// The TransportConn interface represents a transport connection.
// The primary function of a transport connection is to provide the net.Conn interface.
// This interface also exposes access to an underlying network connection,
// which also implements net.Conn.
interface TransportConn extends net.Conn {
  // Conn for the underlying network connection
  networkConn *Conn
}

// The TransportListener interface represents a listener for a transport connection.
// This interface also exposes access to an underlying network listener.
interface TransportListener {
  // Listener for underlying network connection
  networkListener *Listener

  // Accept waits for and returns the next connection to the listener.
  TransportAccept() (TransportConn, error)

  // Close closes the transport listener.
  // Any blocked TransportAccept operations will be unblocked and return errors.
  Close() error
}
```

3.2.4.2. Implementing a Transport

In order to implement a transport, a constructor function must be created that returns an instance of the Transport interface. The transport constructor function, being a normal Go function, can take arbitrary configuration parameters. It is up to the application using the API to implement a valid call to the constructor function for the specific transport being used. This configuration method was chosen over a more generic method such as have a generic constructor method that accepts an associative array or empty interface type as it is more idiomatic to Go. Transports are free to accept basic Go value types as parameters, as well as structs. The Go type system can be used to enforce some validity constraints at compile time.

The Transport instance has two main pieces of functionality: dialing outgoing connections and listening for incoming connections. An instance of the Transport instance must implement a Dial method for making outgoing connections and a Listen method for handling incoming connections. Additionally, it must provide a networkDialer() method to provide access to the network connection dialer that will be used. This allows the application to make changes to how outgoing network connections are made, for instance setting timeouts.

Listening for incoming connections is handled by an instance of the TransportListener interface, which is similar to the standard net.Listener interface. An Accept() function allows for accepting a new incoming transport connection and the Close() function stops listening for incoming connections. The TransportListener wraps a network socket implementer the standard net.Listener interface. The networkListener() function allows access to this network listener, so that the application can configure how incoming network connections are handled.

The transport will also need to implement instances of the TransportConn interface. The Dial and TransportListen functions both return instances of the TransportConn interface. In most cases, these will be different implementations of the interface, one for encoding traffic into the transport's specific protocol and the other for decoding this traffic. TransportConn implements the standard net.Conn interface and will in fact wrap around a standard net.Conn representing the network connection. The networkConn() function returns this underlying network connection so that the application can manipulate the network connection, for instance by setting TCP socket options.

Overall, all network operations are delegated to the transport. For instance, the transport is responsible for initiating outgoing network connections and listening for incoming network connections. This gives the transport flexibility in how it uses the network. Some transports will implement a trivial wrapper, for instance translating Transport.Dial() calls directly into net.Dial() calls. Other transports may implement a more complex relationship between the transport and the network, for instance by using domain fronting.

3.2.4.2.1. Dealing with Configuration Parameters

The configuration of transports is specific to each transport, as each one has different required and optional parameters. The configuration API is therefore also specific to each transport. Each transport provides a constructor function and the type signature for that function specifies the required parameters. For instance, here is an example transport constructor for obfs4:

```
func New(nodeID *ntor.NodeID, publicKey *ntor.PublicKey, sessionKey *ntor.Keypair, iatMode int) *Transport
```

This constructor function provides an idiomatic way to handle configuration. It is the responsibility of the application to handle obtaining the necessary parameters to call the constructor function and to handle deserialization of parameters from any configuration file format used. Each transport may provide helper functions for parsing parameters, but they are not required.

3.2.4.2.2. Wrapping the Network Connection

The transformations provided by each transport to turn data into traffic and back again are provided by the TransportConn implementations returned by the Dial and TransportAccept functions. The TransportConn instances wrap net.Conn instances representing the network connection. A call to TransportConn.Write() will be translated into one or more calls to net.Conn.Write(). Similarly, a call to TransportConn.Read() will be translated into one or more calls to net.Conn.Read(). The underlying network connection is also directly accessible to calling TransportConn.NetworkConn().

3.2.4.3. Using a Transport

Applications using transport have two main responsibilities. The first is gathering transport-specific parameters to pass to the transport constructor function. It is the responsibility of the application to handle obtaining the necessary parameters to call the constructor function and to handle deserialization of parameters from any configuration file format used. Each transport may provide helper functions for parsing parameters, but they are not required. The application must therefore have some understanding of the required parameters for each transport it will use.

The second second responsibility of the application is to set parameters on the network connections underlying the transports. This step is optional and the default network parameters can be used. However, if the application requires more fine-grained control, then then the objects representing the network can be obtained through the Transport.networkDialer(), TransportListener.networkListener(), and TransportConn.networkConn() functions. For instance, the application may need to set the TCP_NODELAY socket option on the network connection in order to disable the Nagle algorithm on the underlying TCP socket.

3.2.5 Javascript Interface

These interfaces are designed to resemble components of the node.js networking API. The interfaces do not collide with one another, so a single object (e.g. a module object returned by `require(‘transport-name’)`) can implement more than one of them.

3.2.5.1. TCP Client Interface

This interface mimics node’s net.createConnection method and the “socks” npm module’s style.

```
interface ClientTransport {
  /** The localStorage and signals parameters are required by complex
transports, and ignored by simple ones. */
  createConnector(configuration:Object, localStorage:string,
signals:Stream) => [Connector];
}
```

```
interface Connector {
  createConnection(settings:Object, options:SocketOptions,
connectListener:function()) => net.Socket;
  name:string;
}
```

3.2.5.2. TCP Server Interface

Similar to client interface, this interface parallels node's net.Server API.

```
interface ServerTransport {
  generateConfiguration() => {client:Object; server:Object;};
  createServer(configuration:Object, local:string, signals:Stream, options,
connectionListener) => net.Server+ServerManager;
  createProxy(configuration:Object, local:string, signals:Stream, options,
connectionListener) => ServerManager;
}

interface ServerManager {
  close(callback);
  getStats() => Object;
}
```

3.2.5.3. UDP Interface

Similar to the client interface, but mimicking the dgram.createSocket API.

```
interface UDPTransport {
  createConnector(configuration:Object, localStorage:string,
signals:Stream) => [UDPConnector]
}
interface UDPConnector {
  /** arguments match node's dgram.createSocket */
  createUDPSocket(options:Object) => dgram.Socket
}
```

3.2.6. Python Interface

This Python API is a proposal. It has not yet been implemented and the current Python implementation of Pluggable Transports, py-obfsproxy, does not follow this API. This draft is primarily intended for discussion about the best choices for an idiomatic API for the Python language.

```
##
#  CLIENT-SIDE

#   On the use side, a transport presents as implementing the same
#   functions as provided by the return value of socket.socket.
#   This can be implemented using an actual socket connecting to another
#   thread.

class TransportFactory:
    """Adds support for one or more transports.  Knows how to either find
       them locally in the process, or how to launch them in a separate
       binary, or find them already running.
    """
    def __init__(self):
        """Initialize empty PluggableTransportSet"""

    def configure(self, configuration):
        """Configure with an opaque configuration string"""

    def getTransportNames(self, transports):
        """Return a list of supplied transport names"""

    def getConnectionFactory(self, transportName):
        """Returns a PluggableTransport for a given transport."""

class PluggableTransport:
    def getTransportName(self):
        """Return the name of this transport"""

    def getProxyConfiguration(self):
        """Return opaque string describing which proxy or proxy-set
           we're using."""

    def configure(self, configuration):
        """Change the configuration for this factory."""

    def makeSocket(self, family, type_=None, protocol=None):
        """Construct a socket-like object.
           Running connect() on this socket will set to destination. The
meaning
           of this destination varies by transport. For transports like obfs4,
           the specified destination is a PT server that speaks the obfs4
           protocol and the Server App address is implicitly already
           known by the PT server. However, for transports like meek, the
           specified destination is the Server App and the PT server
           address is implicitly already known by the transport. Either way,
            the connect() method on the socket-like object returned by
makeSocket
            requires a destination address to be provided.
        """

    def __call__(self, family, type_=None, protocol=None):
        """Drop-in replacement for socket.socket; alias for to makeSocket."""


    def createAsyncTransport(self, asyncLoop):
        """Configure this transport with an asyncio.BaseEventLoop; return an
           AyncPluggableTransport.  Requires that Python has asyncio."""

class AsyncPluggableTransport:
    def create_connection(self, protocol_factory, host=None, port=None, *,
                          ssl=None, family=0, proto=0, flags=0, sock=None,
                          local_addr=None, server_hostname=None):
        """Replacement for BaseEventLoop.create_connection."""

    def create_datagram_endpoint(protocol_factory, local_addr=None,
                                 remote_addr=None, *,
                                 family=0, proto=0, flags=0,
reuse_address=None,
                                 reuse_port=None, allow_broadcast=None,
                                 sock=None):
        """Replacement for BaseEventLoop.create_datagram_endpoint."""


class MetaTransport(asyncio.Transport):
    """
    On the implementation side, a transport must be an asyncio.Protocol with
    the following methods.

    Since asyncio is new in Python 3, this means that to get in-process
    support for a transport, required a Python version >= 3.
    """

    def __init__(self, eventLoop, configuration):
        """Associate this MetaTransport with a single event Loop"""

    def getTransport(self, protocol, onConnect=None):
        """Return an asyncio.Transport object that will pass incoming bytes
           to 'protocol', and handle outgoing bytes.  Inform 'onConnect' when
        the Transport is ready."""
```

3.3 IPC Interface

When the transport runs in a separate process from the application, the two components interact through an IPC interface. The IPC interface serves to ensure compatibility between applications and transports written in different languages.

3.3.1 Pluggable Transport Configuration Environment Variables

When using the IPC interface, Pluggable Transport proxy instances are configured by their parent process at launch time via a set of well defined environment variables and command line flags.

The "TOR_PT_" prefix is used in all environment variable names. This prefix was originally introduced for namespacing reasons and is kept for preserving backwards compatibility with the PT 1.0 specification.

3.3.1.1. Common Configuration Parameters

When launching either a PT Client or PT Server Pluggable Transport, all of the common configuration parameters specified in section 3.3.1.1 MUST be set, using either environment variables or command line flags. Additional configuration parameters specific to PT Clients are specified in section 3.3.1.2 and configuration parameters specific to PT Servers are specified in section 3.3.1.3.

TOR_PT_MANAGED_TRANSPORT_VER or -version
Specifies the versions of the Pluggable Transport specification the parent process supports, delimited by commas. All PTs MUST accept any well-formed list, as long as a compatible version is present.

Valid versions MUST consist entirely of non-whitespace, non-comma printable ASCII characters.

The version of the Pluggable Transport specification as of this document is "2".

Examples
```
TOR_PT_MANAGED_TRANSPORT_VER=1,1a,2,this_is_a_valid_version
obfs4proxy -version 1,1a,2,this_is_a_valid_version
```

TOR_PT_STATE_LOCATION or -state
Specifies an absolute path to a directory where the PT is allowed to store state that will be persisted across invocations. The directory is not required to exist when the PT is launched,

however PT implementations SHOULD be able to create it as required.

If "TOR_PT_STATE_LOCATION" is not specified, PT proxies MUST use the current working directory of the PT process as the state location.

PTs MUST only store files in the path provided, and MUST NOT create or modify files elsewhere on the system.

Examples
```
TOR_PT_STATE_LOCATION=/var/lib/tor/pt_state/
obfs4proxy -state =/var/lib/tor/pt_state/
```

TOR_PT_EXIT_ON_STDIN_CLOSE or -exit-on-stdin-close
Specifies that the parent process will close the PT proxy's standard input (stdin) stream to indicate that the PT proxy should gracefully exit.

PTs MUST NOT treat a closed stdin as a signal to terminate unless this environment variable is set to "1".

PTs SHOULD treat stdin being closed as a signal to gracefully terminate if this environment variable is set to "1".

Example
```
TOR_PT_EXIT_ON_STDIN_CLOSE=1
obfs4proxy -exit-on-stdin-close
```

3.3.1.2. Pluggable PT Client Configuration Parameters

When launching either a PT Client, the common configuration parameters specified in section 3.3.1.1 as well as the client-specific configuration parameters specified in section 3.3.1.2 MUST also be set, using either environment variables or command line flags.

TOR_PT_CLIENT_TRANSPORTS or -transports
Specifies the PT protocols the client proxy should initialize, as a comma separated list of PT names.

PTs SHOULD ignore PT names that it does not recognize.

Parent processes MUST set this environment variable when launching a client-side PT proxy instance.

Example
```
TOR_PT_CLIENT_TRANSPORTS=obfs2,obfs3,obfs4
```

```
obfs4proxy -transports obfs2,obfs3,obfs4
```

TOR_PT_PROXY or -proxy
Specifies an upstream proxy that the PT MUST use when making outgoing network connections. It is a URI [RFC3986] of the format:

<proxy_type>://[<user_name>[:<password>][@]<ip>:<port>.

The "TOR_PT_PROXY" environment variable is OPTIONAL and MUST be omitted if there is no need to connect via an upstream proxy.

Examples
```
TOR_PT_PROXY=socks5://tor:test1234@198.51.100.1:8000
TOR_PT_PROXY=socks4a://198.51.100.2:8001
TOR_PT_PROXY=http://198.51.100.3:443
obfs4proxy -proxy http://198.51.100.3:443
```

3.3.1.3. Pluggable PT Server Environment Variables

When launching either a PT Server, the common configuration parameters specified in section 3.3.1.1 as well as the server-specific configuration parameters specified in section 3.3.1.3 MUST also be set, using either environment variables or command line flags.

TOR_PT_SERVER_TRANSPORTS or -transports
Specifies the PT protocols the server proxy should initialize, as a comma separated list of PT names.

PTs SHOULD ignore PT names that it does not recognize.

Parent processes MUST set this environment variable when launching a server-side PT reverse proxy instance.

Example
```
TOR_PT_SERVER_TRANSPORTS=obfs3,scramblesuit
obfs4proxy - transports obfs3,scramblesuit
```

TOR_PT_SERVER_TRANSPORT_OPTIONS or -options
Specifies per-PT protocol configuration directives, as a semicolon-separated list of <key>:<value> pairs, where <key> is a PT name and <value> is a k=v string value with options that are to be passed to the transport.

Colons, semicolons, equal signs and backslashes MUST be escaped with a backslash.

If there are no arguments that need to be passed to any of PT transport protocols, "TOR_PT_SERVER_TRANSPORT_OPTIONS" MAY be omitted.

Example

```
TOR_PT_SERVER_TRANSPORT_OPTIONS=scramblesuit:key=banana;automata:rule=110;automata:depth=3
Obfs4proxy -options scramblesuit:key=banana;automata:rule=110;automata:depth=3
```

This will pass to 'scramblesuit' the parameter 'key=banana' and to 'automata' the arguments 'rule=110' and 'depth=3'.

TOR_PT_SERVER_BINDADDR or -bindaddr

A comma separated list of <key>-<value> pairs, where <key> is a PT name and <value> is the <address>:<port> on which it should listen for incoming client connections.

The keys holding transport names MUST be in the same order as they appear in "TOR_PT_SERVER_TRANSPORTS".

The <address> MAY be a locally scoped address as long as port forwarding is done externally.

The <address>:<port> combination MUST be an IP address supported by `bind()`, and MUST NOT be a host name.

Applications MUST NOT set more than one <address>:<port> pair per PT name.

If there is no specific <address>:<port> combination to be configured for any transports, "TOR_PT_SERVER_BINDADDR" MAY be omitted.

Example

```
TOR_PT_SERVER_BINDADDR=obfs3-198.51.100.1:1984,scramblesuit-127.0.0.1:4891
obfs4proxy -bindaddr obfs3-198.51.100.1:1984,scramblesuit-127.0.0.1:4891
```

TOR_PT_ORPORT or -orport

Specifies the destination that the PT reverse proxy should forward traffic to after transforming it as appropriate, as an <address>:<port>.

Connections to the destination specified via "TOR_PT_ORPORT" MUST only contain application payload. If the parent process requires the actual source IP address of client connections (or other metadata), it should set "TOR_PT_EXTENDED_SERVER_PORT" instead.

Example

```
TOR_PT_ORPORT=127.0.0.1:9001
obfs4rpxoy -orport 127.0.0.1:9001
```

TOR_PT_EXTENDED_SERVER_PORT or -extorport
Specifies the destination that the PT reverse proxy should forward traffic to, via the Extended ORPort protocol [EXTORPORT] as an <address>:<port>.

The Extended ORPort protocol allows the PT reverse proxy to communicate per-connection metadata such as the PT name and client IP address/port to the parent process.

If the parent process does not support the ExtORPort protocol, it MUST set "TOR_PT_EXTENDED_SERVER_PORT" to an empty string.

Example
```
TOR_PT_EXTENDED_SERVER_PORT=127.0.0.1:4200
obfs4proxy -extorport 127.0.0.1:4200
```

TOR_PT_AUTH_COOKIE_FILE or -authcookie
Specifies an absolute filesystem path to the Extended ORPort authentication cookie, required to communicate with the Extended ORPort specified via "TOR_PT_EXTENDED_SERVER_PORT".

If the parent process is not using the ExtORPort protocol for incoming traffic, "TOR_PT_AUTH_COOKIE_FILE" MUST be omitted.

Example
```
TOR_PT_AUTH_COOKIE_FILE=/var/lib/tor/extended_orport_auth_cookie
obfs4proxy -authcookie /var/lib/tor/extended_orport_auth_cookie
```

3.3.1.4 Command Line Flags

All configuration parameters, including both environment variables and per-connection configuration parameters, can also be provided by using command line flags. When a command line flag is provided, it overrides corresponding environment variables.

3.3.2. Pluggable Transport To Parent Process Communication

When using the IPC method to manage a PT in a separate process, in addition to environment variables and command line flags, a custom protocol is also used to communicate between the application parent process and PT sub-process. This protocol is communicate over the stdin/stdout channel between the processes. This is a text-based, line-based protocol using newline-terminated lines. Lines in the protocol conform to the following grammar:

```
<Line> ::= <Keyword> <OptArgs> <NL>
<Keyword> ::= <KeywordChar> | <Keyword> <KeywordChar>
<KeywordChar> ::= <any US-ASCII alphanumeric, dash, and underscore>
```

```
<OptArgs> ::= <Args>*
<Args> ::= <SP> <ArgChar> | <Args> <ArgChar>
<ArgChar> ::= <any US-ASCII character but NUL or NL>
<SP> ::= <US-ASCII whitespace symbol (32)>
<NL> ::= <US-ASCII newline (line feed) character (10)>
```

The parent process MUST ignore lines received from PT proxies with unknown keywords.

3.3.2.1. Common Messages

IPC messages specified in section 3.3.2.1 are common to both clients and servers.

When a PT proxy first starts up, it must determine which version of the Pluggable Transports Specification to use to configure itself.

It does this via the "TOR_PT_MANAGED_TRANSPORT_VER" (3.2.1) environment variable or -version flag, which contains all of the versions supported by the application.

Upon determining the version to use, or lack thereof, the PT proxy responds with one of two messages: VERSION-ERROR or VERSION.

VERSION-ERROR <ErrorMessage>

The "VERSION-ERROR" message is used to signal that there was no compatible Pluggable Transport Specification version present in the "TOR_PT_MANAGED_TRANSPORT_VER" list.

The <ErrorMessage> SHOULD be set to "no-version" for historical reasons but MAY be set to a useful error message instead.

As this is an error, this message is written to STDERR.

PT proxies MUST terminate with the exit code EX_CONFIG (78) after outputting a "VERSION-ERROR" message.

Examples

```
VERSION-ERROR no-version
```

VERSION <ProtocolVersion>

The "VERSION" message is used to signal the Pluggable Transport Specification version (as in "TOR_PT_MANAGED_TRANSPORT_VER") that the PT proxy will use to configure it's transports and communicate with the parent process.

The version for the environment values and reply messages specified by this document is "2".

PT proxies MUST either report an error and terminate, or output a "VERSION" message before moving on to client/server proxy initialization and configuration.

This message is written to STDOUT.

Examples

```
VERSION 2
```

After version negotiation has been completed the PT proxy must then validate that all of the required environment variables are provided, and that all of the configuration values supplied are well formed.

At any point, if there is an error encountered related to configuration supplied via the environment variables, it MAY respond with an error message and terminate.

ENV-ERROR <ErrorMessage>

The "ENV-ERROR" message is used to signal the PT proxy's failure to parse the configuration environment variables (3.2).

The <ErrorMessage> SHOULD consist of a useful error message that can be used to diagnose and correct the root cause of the failure.

As this is an error, this message is written to STDERR.

PT proxies MUST terminate with error code EX_USAGE (64) after outputting a "ENV-ERROR" message.

Examples

```
ENV-ERROR No TOR_PT_AUTH_COOKIE_FILE when TOR_PT_EXTENDED_SERVER_PORT set
```

3.3.2.2. Pluggable PT Client Messages

IPC messages specified in section 3.3.2.2 are specific to PT clients.

After negotiating the Pluggable Transport Specification version, PT client proxies MUST first validate "TOR_PT_PROXY" (3.2.2) if it is set, before initializing any transports.

Assuming that an upstream proxy is provided, PT client proxies MUST respond with a message indicating that the proxy is valid, supported, and will be used OR a failure message.

PROXY DONE

The "PROXY DONE" message is used to signal the PT proxy's acceptance of the upstream proxy specified by "TOR_PT_PROXY".

This message is written to STDOUT.

PROXY-ERROR <ErrorMessage>

The "PROXY-ERROR" message is used to signal that the upstream proxy is malformed/unsupported or otherwise unusable.

As this is an error, this message is written to STDERR.

PT proxies MUST terminate immediately with error code EX_UNAVAILABLE (69) after outputting a "PROXY-ERROR" message.

Example

```
PROXY-ERROR SOCKS 4 upstream proxies unsupported.
```

After the upstream proxy (if any) is configured, PT clients then iterate over the requested transports in "TOR_PT_CLIENT_TRANSPORTS" and initialize the listeners.

For each transport initialized, the PT proxy reports the listener status back to the parent via messages to stdout and error messages to stderr.

CMETHOD <transport> <'socks4','socks5'> <address:port>

The "CMETHOD" message is used to signal that a requested PT transport has been launched, the protocol which the parent should use to make outgoing connections, and the IP address and port that the PT transport's forward proxy is listening on.

This message is written to STDOUT.

Examples

```
CMETHOD trebuchet socks5 127.0.0.1:19999
```

CMETHOD-ERROR <transport> <ErrorMessage>

The "CMETHOD-ERROR" message is used to signal that requested PT transport was unable to be launched.

As this is an error, this message is written to STDERR.

Outputting a "CMETHOD-ERROR" does not result in termination of the PT process, as even if one transport fails to be initialized, other transports may initialize correctly.

Examples

```
CMETHOD-ERROR trebuchet no rocks available
```

Once all PT transports have been initialized (or have failed), the PT proxy MUST send a final message indicating that it has finished initializing.

CMETHODS DONE

The "CMETHODS DONE" message signals that the PT proxy has finished initializing all of the transports that it is capable of handling.

This message is written to STDOUT.

Upon sending the "CMETHODS DONE" message, the PT proxy initialization is complete.

3.3.2.2.1. Notes

Unknown transports in "TOR_PT_CLIENT_TRANSPORTS" are ignored entirely, and MUST NOT result in a "CMETHOD-ERROR" message. Thus it is entirely possible for a given PT proxy to immediately output "CMETHODS DONE" without outputting any "CMETHOD" or "CMETHOD-ERROR" lines. This does not result in termination of the PT process.

Parent processes MUST handle "CMETHOD"/"CMETHOD-ERROR" messages in any order, regardless of ordering in "TOR_PT_CLIENT_TRANSPORTS".

3.3.2.3. Pluggable PT Server Messages

IPC messages specified in section 3.3.2.3 are specific to PT servers.

PT server reverse proxies iterate over the requested transports in "TOR_PT_CLIENT_TRANSPORTS" and initialize the listeners.

For each transport initialized, the PT proxy reports the listener status back to the parent via messages to stdout and error messages to stderr.

SMETHOD <transport> <address:port> [options]

The "SMETHOD" message is used to signal that a requested PT transport has been launched, the protocol which will be used to handle incoming connections, and the IP address and port that clients should use to reach the reverse-proxy.

This message is written to STDOUT.

If there is a specific <address:port> provided for a given PT transport via "TOR_PT_SERVER_BINDADDR", the transport MUST be initialized using that as the server address.

The OPTIONAL 'options' field is used to pass additional per-transport information back to the parent process.

The currently recognized 'options' are:

ARGS:[<Key>=<Value>,]+[<Key>=<Value>]

The "ARGS" option is used to pass additional key/value formatted information that clients will require to use the reverse proxy.

Equal signs and commas MUST be escaped with a backslash.

Tor: The ARGS are included in the transport line of the Bridge's extra-info document.

Examples
```
SMETHOD trebuchet 198.51.100.1:19999
SMETHOD rot_by_N 198.51.100.1:2323 ARGS:N=13
```

SMETHOD-ERROR <transport> <ErrorMessage>

The "SMETHOD-ERROR" message is used to signal that requested PT transport reverse proxy was unable to be launched.

As this is an error, this message is written to STDERR.

Outputting a "SMETHOD-ERROR" does not result in termination of the PT process, as even if one transport fails to be initialized, other transports may initialize correctly.

Example
```
SMETHOD-ERROR trebuchet no cows available
```

Once all PT transports have been initialized (or have failed), the PT proxy MUST send a final message indicating that it has finished initializing.

SMETHODS DONE

The "SMETHODS DONE" message signals that the PT proxy has finished initializing all of the transports that it is capable of handling.

This message is written to STDOUT.

Upon sending the "SMETHODS DONE" message, the PT proxy initialization is complete.

3.3.3. Pluggable Transport Shutdown

The recommended way for Pluggable Transport using applications and Pluggable Transports to handle graceful shutdown is as follows:

(Parent) Set "TOR_PT_EXIT_ON_STDIN_CLOSE" (3.2.1) when launching the PT proxy, to indicate that stdin will be used for graceful shutdown notification.

(Parent) When the time comes to terminate the PT proxy:

- Close the PT proxy's stdin.
- Wait for a "reasonable" amount of time for the PT to exit.
- Attempt to use OS specific mechanisms to cause graceful PT shutdown (eg: 'SIGTERM')
- Use OS specific mechanisms to force terminate the PT (eg: 'SIGKILL', 'ProccessTerminate()').

PT proxies SHOULD monitor stdin, and exit gracefully when it is closed, if the parent supports that behavior.

PT proxies SHOULD handle OS specific mechanisms to gracefully terminate (eg: Install a signal handler on 'SIGTERM' that causes cleanup and a graceful shutdown if able).

PT proxies SHOULD attempt to detect when the parent has terminated (eg: via detecting that it's parent process ID has changed on U*IX systems), and gracefully terminate.

PT proxies exiting after a graceful shutdown should use exit code EX_OK (0).

3.3.4. Pluggable PT Client Per-Connection Arguments

Certain PT transport protocols require that the client provides per-connection arguments when making outgoing connections. On the server side, this is handled by the "ARGS" optional argument as part of the "SMETHOD" message.

On the client side, arguments are passed via the SOCKS5 protocol authentication mechanism. They keys and values are inserted into a map and this is encoded as JSON.

Example

```
{"shared-secret": "rahasia", "secrets-file": "/tmp/blob"}
```

The arguments are then transmitted when making the outgoing connection using the SOCKS5 authentication mechanism. If no connection settings are present, authentication type 0x00 (no authentication required) is used. If there are connection settings present, authentication type 0x80 (private method) is used, followed by the serialized UTF-8 JSON data.

- The **`CMESSAGE`** and **`SMESSAGE`** commands on STDOUT/STDIN represent messages from client to server and server to client, respectively.

3.3.5 UDP Support

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 Client App will send UDP packets to the PT Client, TCP packets will be sent between the obfuscation and the PT Server, and then the PT Server will send UDP packets to the Server App.

3.3.5.1 Obfuscating Proxy Architecture

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

The architecture of the obfuscating proxy therefore has 4 parts: the Client App, the PT Client, the PT Server, and the Server App. These components are arranged in a bidirectional pipeline where data flows from the Client App, through the pipeline to the Server App, and back again.

3.3.5.2. Configuring the Transports

Each side of the transport (the client and the server) requires certain configuration information in order to function. Many transports require a destination address for the next link in the pipeline. The PT Client may require the address of the PT Server, and likewise the PT Server may require the address of the Server App. However, this is not always required. In the case of domain fronting, for instance, the PT Client chooses the PT Server as part of the domain fronting implementation and so external configuration is not required. Additionally, transport-specific parameters may be required. For instance, the PT Client may require the public key of the PT Server in order to authenticate its identity. Configuration information for PTs is broken up into two types. The first type is a static global configuration provided to when the PT implementation is started. The information is provided by a host process, such as Tor. The host passes the configuration information in through a combination of environment variables and a textual protocol provided through standard input (section 3.3). The second type is per-connection configuration information provided as part of the SOCKS handshake.

In the case of UDP, these configuration mechanisms are missing. The host role is normally provided by Tor, but Tor does not support UDP. Additionally, there is no SOCKS handshake to pass in per-connection configuration information. However, the transports still need all of this configuration information in order to function. In the UDP use case, per-connection configuration information is specified globally with command line flags. The advantage of this approach is that no host process or shell script wrapper is not necessary. The limitation of this approach is that configuration parameters cannot be specified on a per-connection basis.

3.3.5.3. Implementation of the PT Client

The role of the PT Client in UDP mode is to accept UDP packets and relay them over an existing TCP-based transport. The first step is for the PT Client to listen for UDP packets on a designated port. The second step is to relay these packet over a TCP-based transport, which requires two things: a transport connection must be established, and the packets must be converted into a data stream to be written to the transport connection.

Establishing a transport connection requires bridging a mismatch between the semantics of packet-based UDP protocols and connection-based TCP transports. TCP transports are opened and later closed, ending the connection. However, UDP protocols are connectionless. There is no intrinsic way to tell when the first packet will start arrive or when the last packet has arrived. Therefore, the PT Client must establish transport connections using lazy instantiation. The PT Client will maintain a pool of transport connections. Each connection will be associated with a PT Server destination address. When the PT Client receives a UDP packet with a PT Server destination address not represented in the pool, a new transport connection will be created and added to the pool. Otherwise, the existing connection will be used. Additionally, connections will be closed and removed from the pool based on a timeout system. When a connection has not been used for some time, it will be closed. The specific timeout used can be configured. It is also possible that a connection will be closed by the PT Server or due to an error. In this case, the transport will be removed from the pool. The following table shows the state transitions that occur with this implementation.

Event	Current State	New State	Effect
Packet received	No matching Connection in pool	New Connection added to pool with state = Waiting	Packet dropped
Packet received	Matching Connection in pool with state = Waiting		Packet dropped
Packet received	Matching Connection in pool with state = Connected		Packet sent using Connection
Connection successful	Connection in pool with state = Waiting	Connection in pool with state = Connected	
Connection closed	Connection in pool with state = Connected	Remove Connection from pool	
Connection failed	Connection in pool with state = Waiting	Remove Connection from pool	
Write failure sending packet	Connection in pool with state = Connected	Remove Connection from pool	Packet dropped
Timeout since last packet	Matching Connection in pool	Remove Connection from pool	

Table 1. Client-side UDP state transitions

3.3.5.4. Integration with TCP Transports

Configuration of the transports is described in section 3.3.5.2. The remaining integration necessary is to take the receiving UDP packets and convert them to a data stream that can be transmitted over a TCP-based transport connection. The basic mechanism for doing this is described in RFC 5389, “Session Traversal Utilities for NAT (STUN)”. Section 7.2.2, “Sending over TCP or TLS-over-TCP”, describes the necessity for adding additional framing to tell where individual UDP packets start and end within the datastream. The particular implementation of this framing is left unspecified in the RFC.

Two methods of framing can be used. The first is for transparent UDP proxies where the format of the UDP packets is unknown. An example use case for this mode is an OpenVPN proxy. For this mode, a simple two byte length in network byte order can be used to prefix UDP packet payload data. The second mode is specifically for STUN packets. An example use case for this mode is when proxying to a TURN server. As STUN packets already contain a header including a length for the payload, STUN packets can simply be concatenated without additional external framing. Extraction of the individual packets from the data stream on the server side requires knowledge of which framing was used by the client.

3.3.5.5. Implementation of the PT Server

The PT Server receives a data stream over a TCP-based transport connection. It then retrieves the individual packets from the data stream and forwards them on as UDP packets. Two modes of operation are proposed for the PT Server: transparent UDP proxy mode and STUN-aware mode. In the transparent proxy mode, a simple two byte length in network byte order is prefixed to each packet to act as framing metadata. In this mode, packets retrieved from the data stream are forwarded to a destination address specified as a configuration parameter to the PT Server. The STUN-aware mode is similar, except that instead of using external framing metadata, the data stream is treated as a series of STUN packets. The STUN length data is retrieved from the STUN packet headers and used to retrieve the STUN packets. The packets are then forwarded onto a TURN server, the address of which is specified in the PT Server configuration parameters. The goal of the STUN-aware mode is to support the use of existing public TURN servers.

In addition to retrieving packets from the data stream and relaying them onto a UDP Server App, the PT Server must also receive UDP packets from the Server App and relay them back over the transport connection to the PT Client. In this function it follows similar logic to the PT Client. The state transitions possible in the PT Server are similar to those in the PT Client, but there are also differences. If a UDP packet is received and no matching transport connection is available, the packet cannot be delivered and is dropped. Relatedly, connections in the connection pool are always in a Connected state and never in a Waiting state. Therefore Connection states are removed from the state transition table for the PT Server. The following table shows the state transitions that occur with this implementation.

Event	Current State	New State	Effect
Packet received	No matching Connection in pool		Packet dropped
Packet received	Matching Connection in pool		Packet sent using Connection
Connection closed	Connection in pool	Remove Connection from pool	
Write failure sending packet	Connection in pool	Remove Connection from pool	Packet dropped
Timeout since last packet	Matching Connection in pool	Remove Connection from pool	

Table 2. Server-side UDP state transitions

3.3.5.6. Configuring Proxy Modes

There is currently no mechanism for PT Servers to support multiple proxy modes simultaneously. When transport connections are received by the PT Server, the data stream must be interpreted as data from one of the TCP proxy modes (either transparent proxy or SOCKS proxy) or one of the UDP proxy modes (either transparent UDP proxy or STUN-aware proxy to a TURN server). Which mode the PT Server will operate in will be determined by PT Server configuration parameters. It is therefore important to ensure that the PT Client and PT Server are operating in the same mode.

4. Adapters

Adapters enable components written to these different interfaces to work together, to the greatest extent possible.

4.1. Common case: IPC adapters

When an application uses a transport that is written in the same language, it has the option of integrating the transport into its language-specific build system. When they are in different languages, they must communicate through the IPC interface. To minimize duplication of effort, each language could have an *application IPC adapter*, which exposes the language-specific transport interface by wrapping the IPC interface, and a matching *transport IPC adapter*, which does the reverse.

4.2. Special cases

4.2.1. PT 1.0 Compatibility

The PT 2.0 IPC interface for a transport is structurally and functionally similar to the PT 1.0 interface, on both the client and server. Adapters between these interfaces would allow existing application to use new transports, and allow new applications to use existing transports.

4.2.2. Cross-compilation and cross-linking

If two languages are compatible via cross-compilation or cross-language linking, then a suitable adapter (effectively a cross-language “binding”) can enable in-process transport usage.

4.2.3. UDP vs TCP

A TCP transport may be converted into a UDP transport by tunneling UDP inside TCP, using a protocol agreed upon by both endpoints.

4.3. Using the IPC interface in-process

When using a transport that exposes the IPC interface, it may be more convenient to run the transport in a separate thread but in the same process as the application. Packets can still be routed through the transport’s SOCKS5 or TURN port on localhost. However, it may be inconvenient or impossible to use STDIN and STDOUT for communication between these two threads. Therefore, in some languages it may be appropriate to produce an “inter-thread interface” that reproduces the IPC interface’s semantics, but replaces STDIN and STDOUT with language-native function-call and event primitives.

5. Anonymity Considerations

When designing and implementing a Pluggable Transport, care should be taken to preserve the privacy of clients and to avoid leaking personally identifying information.

Examples of client related considerations are:

- Not logging client IP addresses to disk.
- Not leaking DNS addresses except when necessary.
- Ensuring that "TOR_PT_PROXY"'s "fail closed" behavior is implemented correctly.

Additionally, certain obfuscation mechanisms rely on information such as the server IP address/port being confidential, so clients also need to take care to preserve server side information confidential when applicable.

6. References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC1928] Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D., Jones, L., "SOCKS Protocol Version 5", RFC 1928, March 1996.

[EXTORPORT] Kadianakis, G., Mathewson, N., "Extended ORPort and TransportControlPort", Tor Proposal 196, March 2012.

[RFC3986] Berners-Lee, T., Fielding, R., Masinter, L., "Uniform Resource Identifier (URI): Generic Syntax", RFC 3986, January 2005.

[RFC1929] Leech, M., "Username/Password Authentication for SOCKS V5", RFC 1929, March 1996.

7. Acknowledgments

This version of the specification expands upon the PT 1.0 specification.

Appendix A. Example Client Pluggable Transport Session

Environment variables

```
TOR_PT_MANAGED_TRANSPORT_VER=2
TOR_PT_STATE_LOCATION=/var/lib/tor/pt_state/
TOR_PT_EXIT_ON_STDIN_CLOSE=1
TOR_PT_PROXY=socks5://127.0.0.1:8001
TOR_PT_CLIENT_TRANSPORTS=obfs3,obfs4
```

Messages the PT Proxy writes to stdin

```
VERSION 2 PROXY DONE
CMETHOD obfs3 socks5 127.0.0.1:32525
CMETHOD obfs4 socks5 127.0.0.1:37347
CMETHODS DONE
```

Appendix B. Example Server Pluggable Transport Session

Environment variables

```
TOR_PT_MANAGED_TRANSPORT_VER=2
TOR_PT_STATE_LOCATION=/var/lib/tor/pt_state
TOR_PT_EXIT_ON_STDIN_CLOSE=1
TOR_PT_SERVER_TRANSPORTS=obfs3,obfs4 TOR_PT_SERVER_BINDADDR=obfs3-198.51.100.1:1984
```

Messages the PT Proxy writes to stdin

```
VERSION 2
SMETHOD obfs3 198.51.100.1:1984
SMETHOD obfs4 198.51.100.1:43734
ARGS:cert=HszPy3vWfjsESCEOo9ZBkRv6zQ/1mGHzc8arF0y2SpwFr3WhsMu8rK0zyaoyERfbz3ddFw,iat-mode=0
SMETHODS DONE
```