Pluggable Transport Specification

Version 2.0, Draft 3

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. The Transports API (section 3.2) defines a set of language-specific APIs to use transports directly from within an application. A PT library implementing the Transports API is available for the Go language (section 3.2.4). Alternatively, transports can be used through the Dispatcher, a command line tool that runs in a separate process and is controlled through a custom inter-process communication (IPC) protocol. The Dispatcher IPC Interface (section 3.3) provides a way to integrate with applications written in any language and to wrap existing applications in PTs without modifying the source code.

Table of Contents

1. Introduction

This specification describes interfaces for implementing and using Pluggable Transports (PTs). PTs provide a protocol-level mechanism for transforming network traffic between a client application and an intermediary server that applies a reserve transform to the traffic on its way to the destination. This document aims to promote common adoption and easy reuse of PTs for use in anti-censorship tools. Some PTs are focused primarily on obfuscation, whereas others are focused primarily on re-routing traffic via a less-blockable intermediary. Most PTs implement both a PT Client and a PT Server, but in some cases only one of the two is unnecessary.

This document describes two complementary interfaces:

- First, this document describes the “Transport API Interface” and associated implementation details for using and creating pluggable transports that are utilized directly in an application by linking to a library of transport implementations. To use this interface, the application makes calls to library functions to configure the transports and sends and receives data using functions that procide a socket-like interface. This interface is most useful if you are seeking to build a transport directly into your application. It can be the simplest mode of PT integration if the client should be a single binary and should run in a single process.
- Second, this document describes the “Dispatcher IPC Interface.” The dispatcher is a command line tool which the application launches in a separate process. The dispatcher manages the transports, as well as sending and receiving data over the network. The application sends and receives data by communicating with this process. This interface is most useful if the application is written in a different language than the transports or changes to the networking code in the application are not possible. A dispatcher application can in fact use the Transport API Interface internally, serving as a proxy process that the client application will communicate through. This is discussed in more detail in Section 4.1. A dispatcher implementation that wraps the Go implementation of the Transports API is available as a reference implementation [PT2-DISPATCHER].

PTs began as a project of the Tor Project, so Tor is occasionally referenced for backwards-compatibility. However, PTs provide a generic interface for any application to use.

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

The PT Server software exposes a public proxy that accepts connections from PT Clients. The PT Client transforms the traffic before it hits the public Internet and the PT Server reverses this transformation before passing the traffic on to its next destination. By default, the PT Server directly forwards this data to the Server App, but the Server App itself may itself be a proxy server and expect the forwarded traffic it receives to conform to a proxy communication protocol such as SOCKS or TURN. There is also an optional lightweight protocol to facilitate communicating connection metadata that would otherwise be lost such as the source IP address and port [EXTORPORT].

When using the API on both client and server (“Transport API Interface”), the PT Client Library is integrated directly into the Client App and the PT Server Library is integrated directly into the Server App. The Client App and Server App communicate through socket-like APIs, with all communication between them going through the PT library, which only sends transformed traffic over the public Internet.

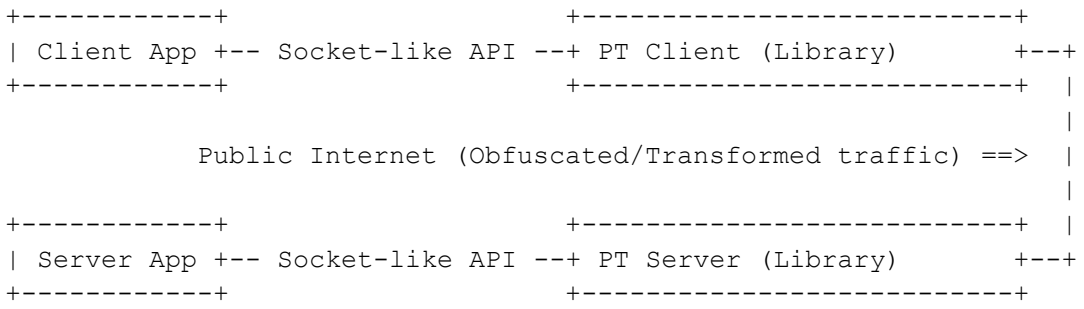

Figure 1. API Architecture Overview

When using the transports as a separate process on both client and server, the Dispatcher IPC Interface is used. On the client device, the PT Client software exposes a local proxy to the client application, and transforms traffic before forwarding it to the PT Server. The PT Dispatcher can be configured to provide different proxy types, supporting proxying of both TCP and UDP traffic.

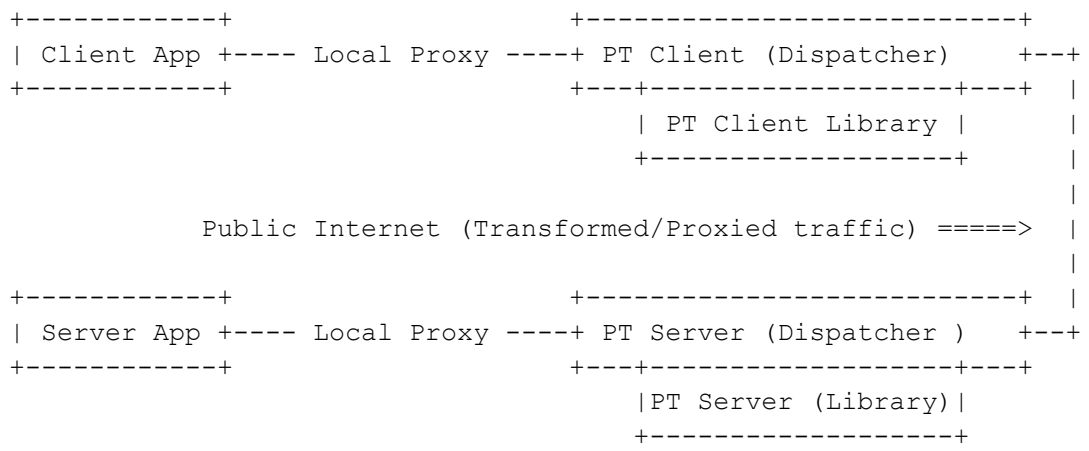

Figure 2. IPC Architecture Overview

A PT may also be function via Dispatcher IPC on one end of the connection but via Transport API on the other, as below (or vice-versa):

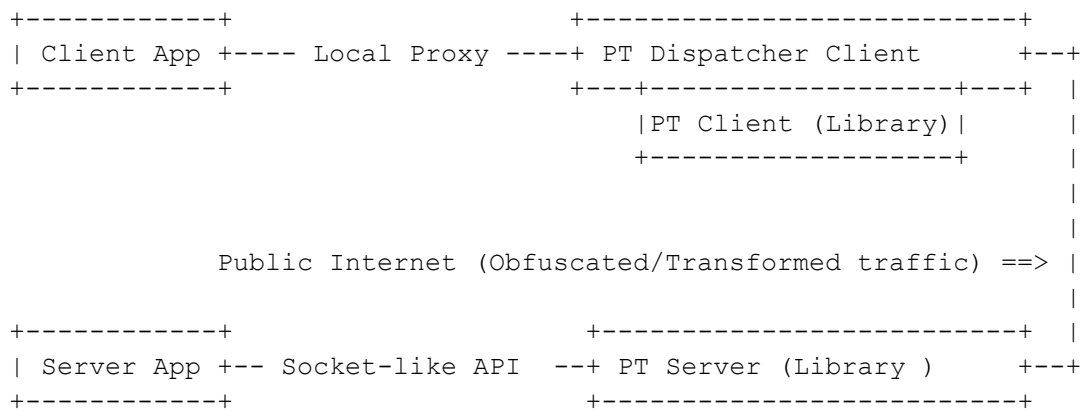

Figure 3. Mixed IPC and Transport API example

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

PT dispatchers 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. Clients SHOULD prefer PT 2.0 over PT 1.0.

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 imposed.

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

3.2. Transports 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 implicitly (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

This section presents high-level pseudocode descriptions of the interfaces exposed by different types of transport components. Implementations for different languages should provide equivalent functionality, but should use the idioms for each language, mimicking the existing networking libraries.

3.2.2.1. Transport

- **Transport** takes a **transport configuration** and provides a **Client Factory** and a **Server Factory**.
    - **Transports** may provide additional language-specific configuration methods
    - The only way to obtain **Client Factories** and **Server Factories** is from the **Transport**.
    - The **Server Factory** of the **Transport** can fail if the Transport does not provide a server-side implementation, such as in the case of the meek transport. However, most transports provide both a client and server implementation.
    - The **transport configuration** is specific to each **Transport**. Using a **Transport** requires knowing the correct parameters to initialize that **Transport**.

3.2.2.1. Client Factory

- **Client Factory** takes the **connection settings** and produces a **Connection** to that server.
    - The **connection settings** are specific to each transport. Some transports will also require an argument indicating the **destination endpoint**. Producing a **Connection** may fail if the server is unreachable or if the **transport configuration** was incorrect.

3.2.2.2. Server Factory

- **Server Factory** takes the address on which the PT server should listen for incoming client connections and produces a **Listener** for that address

3.2.2.2. Listener

- **Listener** produces a stream of **Connections**
    - New **Connections** are available whenever an incoming connection from the PT client has been established. The language-specific API can adopt either a blocking or non-blocking API for accepting new connections, depending on what is idiomatic for the language.

3.2.2.2. Connection

- **Connection** provides an API similar to the environment's native socket type
    - The connection object is extended to provide access to the underlying actual network socket used by the transport, so that low-level networking settings can be changed by the application.

- **Connection** is what is used to read and write data over the transport connection
- The transport-specific logic for obfuscating network traffic is implemented inside the **Connection**.

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 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: base. 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 base

```
package base

// It provides a way to make outgoing transport connections and to accept
// incoming transport connections.
// The Transport interface implements the Transport abstract interface.
type Transport interface {
      // Note that there is no place in this interface to provide the
      // transport configuration. This is provided in the initializer
function
      // for the instance of the Transport interface and so is not included
in
      // the interface definition.

      // Create outgoing transport connection
      // The Dial method implements the Client Factory abstract interface.
      Dial(address string) net.Conn

      // Create listener for incoming transport connection
      // The Listen method implements the Server Factory abstract interface.
      Listen(address string) net.Listener
}

// net.Listener implements the Listener abstract interface.
// This interface is defined in the Go standard library.
```

```
type Listener interface {
	// Accept waits for and returns the next connection to the listener.
	Accept() (net.Conn, error)

	// Close closes the listener.
	Close() error

	// Addr returns the listener's network address.
	Addr() Addr
}

// net.Conn implements the Connection abstract interface.
// This interface is defined in the Go standard library.
type Conn interface {
	// The transport-specific logic for obfuscating network traffic is
	// implemented inside the methods defined in the net.Conn interface.
	//
	// Read reads data from the transport connection. This will likely also
	// require reading data from the underlying network connection. The
	// transport-specific logic for de-obfuscating network traffic is
	// implemented here.
	Read(b []byte) (n int, err error)

	// Write writes data to the connection. This may or may not result in
	// immediate writing of data to the underlying network connection. The
	// transport-specific logic for obfuscating network traffic is
	// implemented here.
	Write(b []byte) (n int, err error)

	// Close closes the transport connection. This will usually also close
	// the underlying network connection used by the transport.
	Close() error

	// These methods are also part of the net.Conn interface. They are not
	// discussed in detail here. For more information on these methods,
look
	// at the official net.Conn documentation.
	LocalAddr() Addr
	RemoteAddr() Addr
	SetDeadline(t time.Time) error
	SetReadDeadline(t time.Time) error
	SetWriteDeadline(t time.Time) 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.

Listening for incoming connections is handled by an instance of the net.Listener interface. An Accept() function allows for accepting a new incoming transport connection and the Close() function stops listening for incoming connections.

The transport will also need to implement instances of the net.Conn interface. The Dial and Listen functions both return instances of the net.Conn 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.

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.

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 net.Conn implementations returned by the Dial and Accept functions. The transport net.Conn instances wrap other net.Conn instances representing the network connection. A call to the transport net.Conn.Write() will be translated into one or more calls to

the network net.Conn.Write(). Similarly, a call to the transport net.Conn.Read() will be translated into one or more calls to the network net.Conn.Read().

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.

3.3 Dispatcher 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 Parameters

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 -ptversion
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 -ptversion 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 on the server or -target on the client
Specifies the destination that the PT reverse proxy should forward traffic to after transforming it as appropriate, as an <address>:<port>. Unless otherwise specified in the documentation of the specific transport being used, the address can be an IPv4 IP address, an IPv6 IP address, or a domain name.

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
obfs4proxy -orport 127.0.0.1:9001
obfs4proxy -target 93.184.216.34:9001
obfs4proxy -target [2001:0db8:85a3:0000:0000:8a2e:0370:7334]:1122
obfs4proxy -target example.com:9922
```

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 communicated 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 is being used, to ensure that it is compatible. It does this via the "TOR_PT_MANAGED_TRANSPORT_VER" (3.2.1) environment variable or -ptversion 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> <'socks5',’transparent-TCP’,’transparent-UDP’,’STUN’> <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 obfs4 socks5 127.0.0.1:19999
CMETHOD meeklite transparent-TCP [::1]:19999
```

```
CMETHOD shadow transparent-UDP [::1]:1234
CMETHOD obfs4 STUN 127.0.0.1:8888
```

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 obfs2 198.51.100.1:19999

SMETHOD obfs4 198.51.100.1:4444
ARGS:cert=60RNHBMRrf+aOSPzSj8bD4ASGyyPl0mkaOUAQsAYljSkFB0G8B8m9fGvGJCp
OxwoXS1baA;iatMode=0

SMETHOD meeklite [2001:0db8:85a3:0000:0000:8a2e:0370:7334]:2323
ARGS:url=https://meek-reflect.appspot.com/;front=www.google.com
```

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', 'TerminateProcess()').

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 Dispatcher IPC protocol. This protocol is based on SOCKS5 and uses the SOCKS5 protocol authentication mechanism. If no per-connection settings are present, authentication type 0x00 (no authentication required) is used.

If there are connection settings present, the authentication type 0x09 (IANA assigned, “JSON Parameter Block”) is used, followed by the serialized per-connection parameter data. The serialization process for the parameters is defined as follows:

- They keys and values are inserted into a map
- This map is serialized JSON to a UTF-8 string.
- The UTF-8 string is converted to a sequence of bytes. (This is trivial for a UTF-8 string.)
- The number of bytes is counted.
- The byte count is encoded as a 4-byte sequence in network byte order (big-endian).
- The encoded count is prepended to the byte sequence.

The following error codes are defined for the response when connection settings are present:

- X’10’ - Connection settings size too large
- X’11’ - Timeout reading connection settings
- X’12’ - Error parsing connection settings
- X’13’ - Connection settings have invalid or missing keys or values

While the byte count is encoded as a 4-byte sequence, which is capable of expressing connection setting sizes up to 4GB, it is not required that the implementation support the maximum possible size. If a size larger than is supported by the implementation is specified, the X’10’ error code can be used. Additionally, an implementation-dependent timeout should included for receiving the connection settings. If this timeout is exceeded, the X’11’ error code can be used. Error code X’12’ is returned if the connection parameters are not properly encoded JSON. Error code X’13’ is used if the connection settings are not correct for the specific transport being used.

Example

```
\x00\x00\x00\x39{"shared-secret": "rahasia", "secrets-file": "/tmp/blob"}
```

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 the PT process when the PT binary 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 neither a host process nor a shell script wrapper is necessary. The PT process can be launched directly from the command line using command line arguments. 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

This section covers the various different ways that the Pluggable Transport interfaces (both API and IPC) can be adapted to different use cases.

4.1. API to IPC Adapter

When an application and the transports it uses are written in the same language, either the Transports API or Dispatcher IPC can be used. When they are in different languages, they must communicate through the Dispatcher IPC interface. For maximum flexibility and to minimize duplication of effort across languages, dispatcher can be implemented by wrapping transport implementations that implement the Transports API. For an example of this approach, see the Shapeshifter Dispatcher [https://github.com/OperatorFoundation/shapeshifter-dispatcher], which wraps transports implementing the Transports API in the Go language and provides a Dispatcher IPC interface to use them from other languages.

4.2. PT 1.0 Compatibility

The only interface defined in the PT 1.0 specification is an IPC interface. No standard API is defined. Therefore, PT 1.0 compatibility refers to compatibility between applications and transports where one side conforms to the PT 1.0 specification and the other conforms to the PT 2.0 specification. Fortunately, an adapter is not needed in this case as both the PT 1.0 and PT 2.0 specifications allow for version negotiation. The TOR_PT_MANAGED_TRANSPORT_VER environment variable or -ptversion command line flag is used by the application to specify a list of supported versions, for instance “1.0,2.0”. The PT provider responds with the VERSION command on stdout in order to specify which version is supported by the PT provider, for instance “VERSION 2.0”. Since the application can specify a list of supported versions, the PT provider can respond dynamically, supporting PT 1.0 when required and automatically upgrading to a PT 2.0 implementation when that is an available option. It is up to applications whether they want to support PT 2.0 exclusively or maintain backwards compatibility with PT 1.0 implementations.

4.3. Cross-language Linking

If two languages are compatible via cross-language linking, then a suitable adapter can be written that wraps the implementation of the Transports API in one language with an API for a compatible language. For example, on Android the Go implementation of the Transports API is wrapped in a Java API to create Java language bindings without the need for a native Java implementation or use of Dispatcher IPC.

4.4. Using the Dispatcher IPC Interface In-process

When using a transport that exposes the Dispatcher 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 Dispatcher IPC interface’s semantics, but replaces STDIN and STDOUT with language-native function-call and event primitives. This is the approach used by OnionBrowser [https://mike.tig.as/onionbrowser/], the Tor implementation on iOS. This approach is used because Tor uses the Dispatcher IPC mechanism to talk to the transports instead of the Transports API. However, iOS does not allow for applications to have multiple processes. Therefore, an in-process Dispatcher IPC approach must be used instead of traditional separate process Dispatcher IPC. An alternative would be to use the Transports API directly instead of Dispatcher IPC.

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 and 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.

[PT2-DISPATCHER] Wiley, Brandon., Shapeshifter Dispatcher. https://github.com/OperatorFoundation/shapeshifter-dispatcher

7. Acknowledgments

Many people contributed to the PT 2.0 specification. Major contributions were made by Dr. Brandon Wiley (Operator Foundation), Nick Mathewson (Tor), and Ben Schwartz (Jigsaw). Valuable feedback was provided by the attendees at the Pluggable Transport Implementers Meetings and the traffic-obf and tor-dev mailing lists. The PT 2.0 specification expands upon the “Pluggable Transport Specification (Version 1)” document authored by Yawning Angel (Tor). Inspiration for the PT 2.0 Go API was also inspired by the obfs4proxy implementation of the PT 1.0 specification in Go, also developed by Yawning Angel (Tor).

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
```

Appendix C. Changelog

Draft 3

- Expanded acknowledgements section
- Removed TransportConn and TransportListener in favor of net.Conn and net.Listener
- Modified SOCKS authentication method to use IANA-assigned designator
- Added error response codes for per-connection arguments

Draft 2

- Renamed version flag to ptversion to avoid naming conflict with goptlib
- Modified Go examples to use correct Go syntax
- Renamed pt module in Go examples to base to avoid naming conflict with goptlib
- Reworded introduction
- Clarified Go examples with more details on how to implement a transport in Go
- Removed unused Javascript and Python APIs
- Removed SSH transport example
- Standardized use of Transports API and Dispatcher IPC language throughout
- Added length to per-connection parameter encoding