Pluggable Transport Base Specification

Version 3.0

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.

This specification consists of several documents, each with its own independently tracked version number. The PT specification as a whole can be considered to be the collection of the current set of documents. The version numbers that makes up the PT 3.0 specification are included in this overview document.

There are two ways to use transports. The Transport API defines a set of language-specific APIs to use transports directly from within an application. PT libraries implementing the Transport APIs for the Go, Swift, and Java language are available. 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 specification 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
    1.1. Requirements Notation

2. Architecture Overview

3. Specification
    3.1. Pluggable Transport Naming
    3.2. Transports API Interface
        3.2.1. Goals for interface design
        3.2.2. Abstract Interfaces
            3.2.2.1. Transport
            3.2.2.1. Client Factory
            3.2.2.2. Server Factory
            3.2.2.2. Listener
            3.2.2.2. Connection

4. Adapters
    4.1. API to IPC Adapter
    4.2. PT 1.0 Compatibility
    4.3. Cross-language Linking
    4.4. Using the Dispatcher IPC Interface In-process

5. Anonymity Considerations

6. References

7. Acknowledgments

Appendix A. Example Client Pluggable Transport Session

Appendix B. Example Server Pluggable Transport Session

Appendix C. Changelog

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.

The PT 3.0 specification consists of the following documents:

- Base specification (this document) - v3.0
- Go API - v3.0
- Swift API - v2.0
- Java API - v1.0
- IPC protocol - v3.0

The specification describes two complementary interfaces:

- First, the specification 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 provide 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. There are two API interfaces specified in PT 2.1, for the Go and Swift programming languages.
- Second, the specification 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].

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.

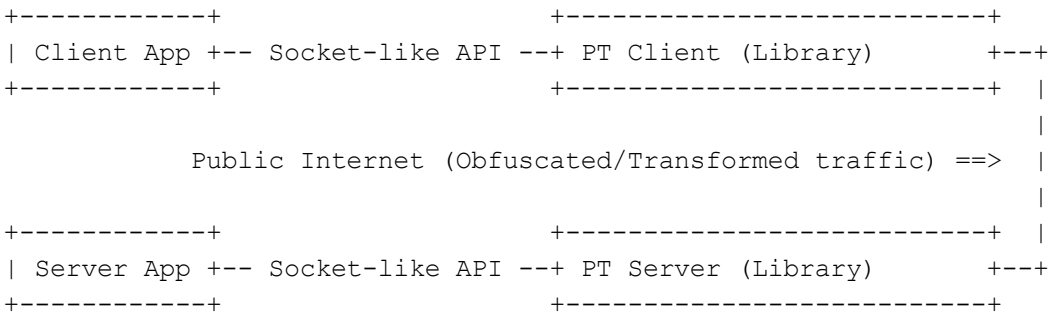

Figure 1. API Architecture Overview

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.

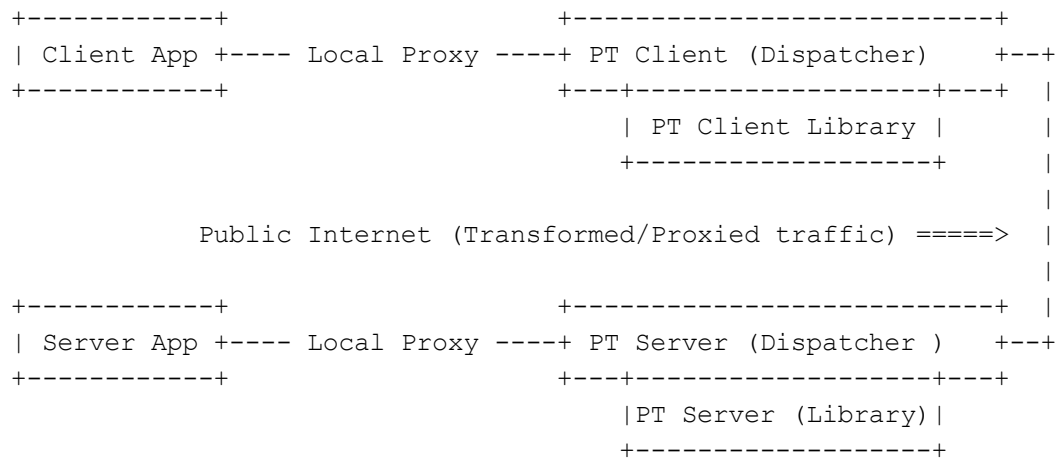

Figure 2. IPC Architecture Overview

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):

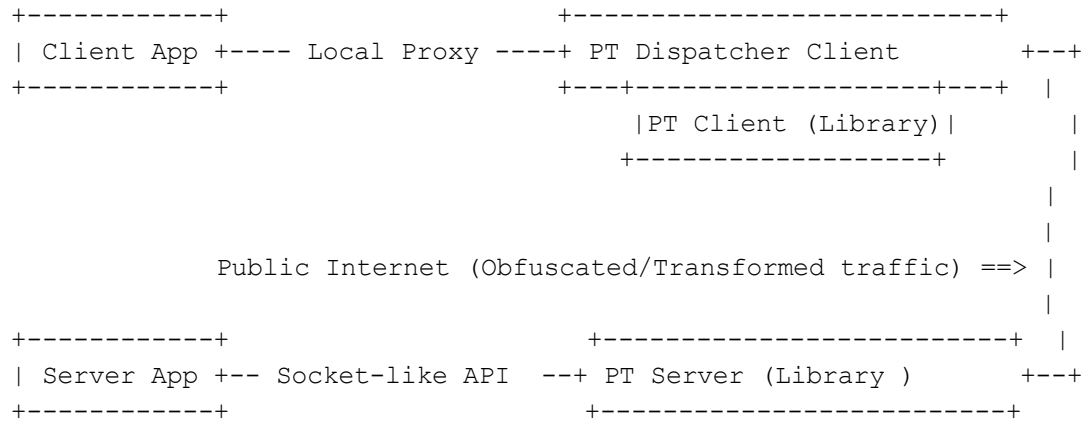

Figure 3. Mixed IPC and Transport API example

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

PT 3.0 dispatchers MAY support any of the following proxy modes: transparent-TCP, transparent-UDP, socks5, and STUN.

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 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 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.
    - 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 **client settings** and produces a **Socket-like Object** that can be used to send and receive data. Both connection-oriented data streams and connectionless messages are available interfaces, similar to TCP and UDP sockets. Transports can support either interface or both.
    - The **connection settings** are specific to each transport. Producing a **Socket-like Object** may fail if the **transport configuration** was incorrect or due to environmental factors such as unreachable servers.

3.2.2.2. Server Factory

- **Server Factory** takes the **server settings** and produces a **Socket-like Object** which listens for incoming data from clients. Both connection-oriented data streams and connectionless messages are available interfaces, similar to TCP and UDP sockets. Transports can support either interface or both. For connection-oriented data streams, the **Server Factory** produces **Listeners** which can be used to obtain **Connections** to clients. For connectionless interfaces, messages can be retrieved directly.

3.2.2.2. Listener

- **Listener** is a type of **Socket-like Object** that produces a stream of **Connections**
    - For connection-oriented data stream interfaces, 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.
    - For connectionless message-oriented interfaces, a **Listener** is not needed.

3.2.2.2. Connection

- **Connection** is a type of **Socket-like Object** that 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.
- For connection-oriented data stream interfaces, a **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**.
- For connectionless message-oriented interfaces, a **Connection** is not needed. Messages can be sent directly without first establishing a **Connection**.

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. IPC Version Compatibility

The PT 3.0 IPC protocol is not compatible with the PT 2.0 or PT 1.0 versions. However, an implementation of the specification can be mutually compatible with multiple versions of the protocol through version negotiation.

For PT 3.0, the -ptversion flag is used to specify a list of versions supported by the host application. For PT 1.0, the TOR_PT_MANAGED_TRANSPORT_VER environment variable serves the same purpose. For PT 2.0, either can be used. Therefore, an application wanting to be compatible with all versions should first check the -ptversion flag and, if it is not present, check the TOR_PT_MANAGED_TRANSPORT_VER environment variable. Having retrieved the list of versions and having determined if there is a version of the protocol that is compatible with by the host application and the dispatcher, the dispatcher responds with the VERSION command on stdout in order to specify which version is supported by the PT provider, for instance “VERSION 3.0”. Since the application can specify a list of supported versions, the PT provider can respond dynamically, supporting PT 1.0 or PT 2.0 when required and automatically

upgrading to a PT 3.0 implementation when that is an available option. It is up to applications whether they want to support PT 3.0 exclusively or maintain backwards compatibility with PT 2.0 and 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, while there is currently no C API for PTs, the Go API has been successfully exported through CGo facilities and wrapped in a C API for integration into the OpenVPN codebase, which is written in C.

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/] on iOS. This approach is used because OnionBrowser 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 the -proxy "fail closed" behavior is implemented correctly.

Additionally, certain obfuscation mechanisms rely on configuration parameters 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] Operator Foundation, Shapeshifter Dispatcher. https://github.com/OperatorFoundation/shapeshifter-dispatcher

7. Acknowledgments

Many people contributed to the PT 3.0 specification. In particular, the Pluggable Transport Steering Committee and the attendees at the Pluggable Transport Implementers Meeting provided many helpful suggestions and feedback. The PT 3.0 specification is a refinement of the PT 2.0 specification and draws most of its text from those documents.

Appendix A. Changelog

PT 3.0

- Implemented proposal 0007 - Java API
- Implemented proposal 0014 - Make Server Address Part of Client Options
- Removed language assuming that transports and applications will use connection-oriented data streams

PT 2.1

- Implemented proposal 0002 - Modularization of Specification

PT 2.0, Draft 3

- Expanded acknowledgements section

PT 2.0, Draft 2

- Reworded introduction
- Removed unused Javascript and Python APIs
- Removed SSH transport example
- Standardized use of Transports API and Dispatcher IPC language throughout