Pluggable Transports Work In Progress

Discussion document for the Valencia workshop

These ideas are current as of 2016 February 25

Concept

Pluggable transports 1.0 is dominated by a single implementation (obfs4proxy) and a single client-application (Tor). For PT 2.0, we want to encourage a wider variety of independent transport implementations, and also meet the needs of a wider variety of applications.

Different transport implementations may most naturally provide different functionality, depending on their purpose. Different applications may most naturally consume different interfaces, depending on their needs.

Our goal for PT 2.0 should therefore be to define multiple distinct interfaces for supplying or accessing pluggable transport services. The PT community may then share “adapter” code that converts one interface into another, so that each application and transport may be written to its most natural interface, with a minimum of additional work.

This library of adapters will form the “implementation” of the PT 2.0 specification. Together with the documented interfaces, these adapters should enable software authors to write clients and transports that interoperate with the rest of the PT 2.0 ecosystem.

Non-goals

This document does not explain how to provide a directory of pluggable transports, or how to plug in new transports at runtime. That is an exciting possibility, but this document focuses on “compile-time” linking of transports into applications.

Some PT servers may employ a whitelist or blacklist of allowed destinations (e.g. only Tor nodes, only a specific localhost service). Ensuring that this restriction is appropriate for the application is outside the scope of this document.

Goals for interface design

We seek interfaces with 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.
- Transport servers can either be *bound* to a specific service or *free* to proxy connections to a variety of destinations, with minimal changes to client, server, user, or protocol.
- 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 transport 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.

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.

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.
    - If the transport is *free*, then the ConnectionFactory also requires an argument indicating the TCP **destination endpoint**.
    - The connection object is extended to have an additional **get stats** method

Advanced 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).

TCP Server

Similar to the TCP client, there is a basic interface for stateless, non-negotiating clients, and a more advanced 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)
    - TODO: What about forwarding a single local port?
  - 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

Advanced 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).

UDP Peer

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

Examples

SSH Key-based Free Transport

Consider a TCP transport for clients that uses SSH as the transport. This is a free transport, allowing connections to all hosts. 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.

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

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.

IPC Interfaces

When the transport runs in a separate process from the application, the two components interact through an IPC interface that provides equivalent functionality to the abstract interface. The IPC interface serves to ensure compatibility between applications and transports written in different languages. It also enables runtime-pluggable transports, and may be helpful when using blocking socket APIs in languages that lack easy multithreading.

TCP Client and Server interfaces

When implementing TCP functionality in an IPC interface, we recommend a variant of SOCKS5. This interface is similar to the PT 1.0 IPC interface except

- Instead of username/password authentication type, clients use
    - type 0x00 (no authentication required) if the **connection settings** are empty/null
    - otherwise type 0x80 (private method), following by **connection settings** serialized to UTF-8 JSON
- In a *bound transport*, no TCP destination endpoint is specified, so the SOCKS destination field is set to [::0]:0, i.e. port zero on the all-zeros IPv6 address.
- Instead of environment variables, the **configuration** is encoded as JSON and passed as a command-line argument to the executable
- Error messages are directed to STDERR
    - Fatal errors are also returned as nonzero exit codes following the sysexits standard
- State should be written to files only in the child process's current directory
- The **`CMESSAGE`** and **`SMESSAGE`** commands on STDOUT/STDIN represent messages from client to server and server to client, respectively.

UDP Interface

For IPC connections, transports should implement a compliant TURN server, implementing RFC 5766 (but TCP support is not required). As with the TCP client and server, **configuration** is passed as a JSON command-line argument, and metadata is passed between application and transport on STDIN and STDOUT. **Connection settings** are not supported (UDP is connectionless!).

Javascript Interfaces

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.

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 {
  isBound() => boolean;
  /**
```

```
   * If isBound() returns true, then this connector has a fixed
   * destination.  Otherwise, |options| must indicate the destination host
   * and port.
   */
  createConnection(settings:Object, options:SocketOptions,
connectListener:function()) => net.Socket;
  name:string;
}
```

Open questions:

- |localStorage| must be a string (path) to wrap the IPC interface, but for a transport implementer a higher-level object (e.g. HTML5 LocalStorage) might be more appropriate.

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;
}
```

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
}
```

Python interface

```
##
#  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 isBound(self):
        """Return true iff the sockets made with this factory cannot connect
           to any address other than a hard-wired destination."""
```

```
    def makeSocket(self, family, type_=None, protocol=None):
        """Construct a socket-like object.  If this is a bound transport, then
           running connect() on this socket will set what PT server to
           connect to.  If this is a free transport, then connect() on
           the socket will set the final destination to connect to,
           and the PT server has already been configured.
        """

    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, you need to be running a new-ish python.
    I think we're okay with that.)
    """
```

```
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."""
```

Adapters

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

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 will 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.

These IPC adapters also allow applications to support plugging in new transports at runtime.

Special cases

PT 1.0 Compatibility

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

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. This is likely to be relevant when using Go transports in Javascript applications.

Bound vs. Free

Bound and free transports can be interconverted using appropriate adapters. However, converting a bound transport into a free transport requires multiplexing several streams through

a single transport using a protocol not specified here. Therefore, a compatible adapter is required on both client and server.

UDP vs TCP

A UDP or TCP transport is defined by the protocol it *carries*, not the protocol it *uses*. A TCP transport may therefore be converted into a UDP transport by tunneling UDP inside TCP, using a protocol agreed upon by both endpoints.

Future Interfaces

Bytestreams

Authors wishing to create simple experimental transports might be deterred by the difficulty of implementing not only the data transformation, but also all of the networking logic required to establish a connection. To help transport authors in this situation, we should offer a simple bytestream encoding/decoding interface for transports.

This interface should support backpressure and Nagle-style delayed output, but may not support time-based actions from the transport itself.

In Javascript, it would be logical to adopt the Web Streams API or the node streams API (a.k.a. streams2) for TCP stream processing. In systems languages, the logical interface might be an abstract socket pair (e.g. io.BufferedRWPair in Python).

For UDP processing, the corresponding abstraction is a packet stream. In Javascript, uProxy has already developed an interface for this purpose. This interface is N:M for packets in and out, and is symmetrical (no client/server distinction).

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.