Swift Transport API

Version 1.0, Draft 1

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

The Transport APIs defines a set of language-specific APIs to use transports directly from within an application. This document describes the Transport API for the Swift programming language, as well as associated implementation details for using and creating pluggable transports that are utilized directly in a Swift application or Network Extension 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 which is also written in the Swift programming language. It can be the simplest mode of PT integration if the client should be a single binary and should run in a single process. It is also designed to work well with macOS and iOS Network Extensions, which is the way that Apple has provided for providing VPN services on iOS and macOS.

Table of Contents

[1. Swift Transport API]

- [1.1 Client Constructor Functions]
- [1.2 Interfaces for TCP connections and UDP sessions]
- [1.3 Extensions for compatibility with Apple’s Network API]
- [1.4. Implementing a Transport]
  - [1.4.1. Dealing with Configuration Parameters]
  - [1.4.2. Wrapping the Network Connection]
- [1.5. Using a Transport]

[Appendix A. Changelog]

1. Swift Transport API

There are many ways to do networking on iOS and macOS, from traditional BSD sockets to high-level abstractions such as URLSession. The official recommendation from Apple for doing TCP and UDP connections is to use the Network framework. It is important to differentiate between the Network framework, which provides an API for making TCP and UDP network connections and the Network Extension framework, which is for managing VPNs. In the case of Pluggable Transports, both of these frameworks come into play as Swift Pluggable Transports may be running inside of a Network Extension, but use the Network framework to connect to the transport server. There are a variety of classes provided by the Network framework, such as NWConnection to represent network connections and NWEndpoint to represent network endpoints such as servers. These classes can be used either from inside of a normal application or inside of a Network Extension.

The Swift API for transports provides a set of interfaces and implementations that mimic those found in the Network framework. Modifying code from using the Network framework directly to using transports is a simple matter of renaming the types. For instance, instead of using the NWConnection class, use the Connection interface. Additionally, the Swift API provides extensions that make the Network framework conform to the transport API. For instance, the NWConnection class has been extended to conform to the Connection interface. Therefore, the type Connection can be used everywhere in the code, whether the code is using Pluggable Transports or directly connecting to the network using the Network framework.

1.1 Client Constructor Functions

A transport is implemented by providing an implementation of the Connection interface. This requires specifying an init() function to create new instances. However, when nesting transports it may be necessary to create multiple connections over time using a provided Connection type. In order to support this behavior, a client connection factory must be provided. Here are some example connection factories:

```
protocol ConnectionFactory {
      connect(using: NWParameters) -> Connection?
}
```

This is the definition of the ClientFactory interface that transports must provide.

```
class MeekConnectionFactory: ConnectionFactory
{
      init(to: URL, serverURL: URL)
      init(to: URL, serverURL: URL, factory: ConnectionFactory)
      connect(using: NWParameters) -> Connection?
```

```
}
```

This first init() is a basic initializer for the Meek transport. It takes transport-specific arguments, in this case the “to” and “serverURL” parameters. The connect() method can then be called multiple times to create connections. The connect method takes an NWParameters argument to specify information related to the Network framework. For instance, this parameter is how you determine whether the returned connection is a TCP or UDP socket.

The second init() allows for the Meek transport to take an existing ConnectionFactory to use to make its connection to the network. This allows for nesting of Meek with other transports. If no factory is provided, Meek will use the standard NWConnection provided by the Network framework to connect to the network.

```
class WispConnectionFactory
{
	init(to: NWEndpoint, cert: String, iatMode: Bool) ->
Connection?
	init(to: NWEndpoint, cert: String, iatMode: Bool,
		factory: ConnectionFactory)
	init(to: NWEndpoint, cert: String, iatMode: Bool,
		connection: Connection)
	connect(using: NWParameters) -> Connection?
}
```

This first init() is a basic initializer for the Wisp transport. It takes transport-specific arguments, in this case the “to”, “cert”, and “iatMode” parameters. The connect() method can then be called multiple times to create connections. The connect method takes an NWParameters argument to specify information related to the Network framework. For instance, this parameter is how you determine whether the returned connection is a TCP or UDP socket.

The second init() allows for the Wisp transport to take an existing ConnectionFactory to use to make its connection to the network. This allows for nesting of Wisp with other transports. If no factory is provided, Wisp will use the standard NWConnection provided by the Network framework to connect to the network.

The third init() allows for the Wisp transport to take an already established Connection to use to as its connection to the network. This allows for nesting of Wisp with other transports, but if the connection fails then Wisp will have no way of making another connection because it does not have access to a ConnectionFactory. The advantage of this approach is that it gives the application developer greater control of how the underlying network connection is formed, for transports that support this initializer. If no connection is provided, Wisp will internally create a new standard NWConnection provided by the Network framework to connect to the network.

As shown in the examples above, each constructor function can take different arguments, including the configuration parameters for the specific transport. Transports can also have multiple constructor functions. Transport constructor functions return custom connection objects which implement the NWConnection protocol. Optionally, transports can accept as arguments to their constructor functions either a ConnectionFactory or a Connection. As each transport connect() function also returns an instance of Connection, this allows transports to be nested.

The difference between taking a ConnectionFactory or a Connection in the constructor function is that a ConnectionFactory allows the transport to create multiple connections, whereas accepting a Connection only allows the transport to use the existing connection.

Ultimately, at the bottom of any stack of transports, there is an underlying Connection which is an instance of NWConnection, a class provided by the Network framework. While interchangeable with Connections provided by transports, this Connection represents the real connection over the network. Transport Connections stack on top of the NWConnection to provide transformations on the network traffic.

1.2 Interfaces for TCP connections and UDP sessions

Transport constructor functions return custom connection objects which implement the Connection protocol. This protocol provides a subset of the functionality of the Network Extension framework's NWConnection class.

```
public protocol Connection
{
      func start(queue: DispatchQueue)
      func cancel()

      var stateUpdateHandler: ((NWConnection.State) -> Void)?
      var viabilityUpdateHandler: ((Bool) -> Void)?

      func send(content: Data?,
            contentContext: NWConnection.ContentContext = default,
            isComplete: Bool = default,
            completion: NWConnection.SendCompletion)

      func receive(completion: @escaping (
                  Data?, NWConnection.ContentContext?,
                  Bool,
                  NWError?
            ) -> Void)

      func receive(minimumIncompleteLength: Int,
```

```
            maximumLength: Int,
            completion: @escaping (
                  Data?, NWConnection.ContentContext?, Bool, NWError?
            ) -> Void)
}
```

The start() and cancel() methods are used to start and stop the connection.
The stateUpdateHandler and viabilityUpdateHandler variables hold callback functions which are used to get updates on when the connection state has changed. Viability represents whether the connection is up and ready to use or not, whereas state provides more fine-grained information.
The send() and two different receive() methods are used to send and receive data of the connection, respectively. Callback functions are specified to allow the application to be notified when the send or read operations are finished.

Optionally, transports can also support applications that communicate with UDP packets. The ConnectionFactory provides a unified interface to both TCP and UDP applications. When the application calls connect() on the ConnectionFactory, it provides an NWParameters. Among many other options, this also specifies whether the returned Connection should provide TCP or UDP semantics. Transports that do not support Connections conforming to the requested type as specified in the NWParameters should return nil instead of a Connection instance.

1.3 Extensions for compatibility with Apple’s Network API

The Network framework is not extensible because it uses classes for types. Since these classes cannot be subclassed, there is no way to provide an independent implementation of Network Extension classes such as NWConnection. Therefore, transports cannot be used directly with code that expects a Network framework NWConnection. However, the Swift API for Pluggable Transports is more flexible because it uses interfaces for types instead of classes. Networking code in an application needs to be migrated from using Network framework types such as NWConnection to Swift Transport API types such as Connection. Once this is done, the networking code can use either Network framework connections or transport connections interchangeably. The Swift Transport API accomplishes this by providing a type-compatible wrapper around the Network framework classes.

1.4. Implementing a Transport

In order to implement a transport, a constructor function must be created that returns an instance of the Connection interface. The transport constructor 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.

The transport will also need to implement the other methods necessary to fulfill the requirements of the Connection protocol. In particular, there are methods for sending and receiving data. There are also callbacks that must be handled for dealing with connection state.

Overall, all network operations are delegated to the transport. For instance, the transport is responsible for initiating outgoing network connections. This gives the transport flexibility in how it uses the network.

1.4.1. 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:

```
init(to: NWEndpoint, cert: String, iatMode: Bool,
          connection: Connection)
```

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.

1.4.2. Wrapping the Network Connection

The transformations provided by each transport to turn data into traffic and back again are provided by the Connection implementations returned by the constructor functions. The transport Connection instances wrap other Connection instances representing the network connection. A call to the transport Connection.send() will be translated into one or more calls to the network Connection.send(). Similarly, a call to the transport Connection.receive() will be translated into one or more calls to the network Connection.receive().

1.5. 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.

Appendix A. Changelog

PT 2.1, Draft 1

- First draft