(This document is based on the following template: https://goo.gl/BZdVhM)

Swift API

Proposal: #3
Authors: Dr. Brandon Wiley, Operator Foundation
Status: Awaiting completion of implementation

Implementation: Work in progress: https://github.com/OperatorFoundation/Transport

Introduction

This proposal details an API for implementing transports in the Swift programming language, which is used on iOS and macOS. The primary goal of providing a Swift implementation is to allow transports to work inside of a Network Extension, which is the way that Apple has provided for providing VPN services on iOS and macOS.

Motivation

A Go API for transports already exists and compiles on several platforms, including macOS and iOS. However, recently Apple has introduced a new way to provide VPN services on macOS and iOS called a Network Extension. A Network Extension runs separately from applications and handles network communication. It is the official way supported by Apple to provide VPN services on macOS and iOS. Unfortunately, code written in Go has trouble running inside of a Network Extension because Apple severely limits the memory available and the Go runtime requires more memory than is available.

One possible solution to this problem is to provide a Swift implementation of transports, with the most popular transports being ported from Go to Swift. The Go implementation remains the reference implementation for most transports and the Go implementation can still be used on the server side. On the client side of the transport needs to be ported to Swift, and only for the most popular transports which are worth the efforting of providing for iOS and macOS, in addition to all of the other platforms where the Go implementation already runs well.

In order to write the Swift implementations, a Swift API for transports must first be defined. Since the goal of the Swift implementation is to enable transports to function inside of a Network Extension, this should be taken into account in the design of the Swift API.

The proposed solution seeks to meet the following goals in defining a Swift API:

- Provide an API similar to the existing API for networking

- Work inside of a Network Extension
- Allow for transport of both TCP and UDP application traffic
- Allow stacking of transports (one transport can run on top of another)
- Work with existing popular transports such as meek and obfs4
- Work with both IPv4 and IPv6 addresses

Proposed solution

The proposed Swift API consists of the following parts:

- Constructor functions for making connections
- Interfaces for representing TCP connections and UDP sessions
- Extensions for compatibility with Apple's Network Extension API

Design

There are many ways to do networking on iOS and macOS, from traditional BSD sockets to high-level abstractions such as URLSession. However, doing networking inside of a Network Extension uses none of these and instead defines its own networking API defined in the NetworkExtension framework. There are a variety of classes provided, such as NEPacketTunnelProvider and NWTCPConnection. As Network Extensions are not applications, they do not have access to the normal APIs used by applications, such as BSD sockets. All networking must be done through the NEPacketTunnelProvider, using the APIs it makes available. It is important to note that this restriction is on the code running inside of the Network Extension and not on the application using the Network Extension. Applications can use the normal networking APIs, such as BSD sockets or URLSession. When a Network Extension is running, the application's networking calls are made to a virtual network device. The operating system intercepts all network traffic and gives it to the Network Extension for processing. The Network Extension is therefore tasked with making the real network connection, which it must do using the NEPacketTunnelProvider.

The Swift API for transports provides a set up interfaces and constructor functions that mimic those found in the NetworkExtension framework. Modifying code from using the NetworkExtension framework directly to using transports is a simple matter of renaming the types. For instance, NEPacketTunnelProvider becomes PacketTunnelProvider and NWTCPConnection becomes TCPConnection. Additionally, the Swift API provides extensions that make the NetworkExtension framework conform to the transport API. For instance, an NEPacketTunnelProvider is extended to be an instance of the PacketTunnelProvider interface. Therefore, the type PacketTunnelProvider can be used everywhere in the code, whether a transport or the system tunnel provider is being used.

Client Constructor Functions

A transport is implemented by providing a constructor function. Here are some example constructor functions:

```
func createMeekTCPConnection(
      provider: PacketTunnelProvider,
      to: URL,
      serverURL: URL
) -> MeekTCPConnection?

func createWispTCPConnection(
      provider: PacketTunnelProvider,
      to: NWEndpoint,
      cert: String,
      iatMode: Bool
) -> WispTCPConnection?

func createWispTCPConnection(
      connection: TCPConnection,
      cert: String,
      iatMode: Bool
) -> WispTCPConnection?
```

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 TCPConnection protocol. Optionally, transports can accept as arguments to their constructor functions either a PacketTunnelProvider or a TCPConnection. As each transport constructor function also returns an instance of TCPConnection, this allows transports to be nested.

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

The PacketTunnelProvider is an interface wrapping a generic constructor function. This interface is defined as follows:

```
public protocol PacketTunnelProvider {
      func createTCPConnectionThroughTunnel(
            to remoteEndpoint: NWEndpoint,
```

```
            enableTLS: Bool,
            tlsParameters TLSParameters: NWTLSParameters?,
            delegate: Any?
      ) -> TCPConnection?
}
```

Transports are only required to provide a constructor function. They are not required to provide an implementation of PacketTunnelProvider. However, if they do then this allows other transport with constructor functions accepting a PacketTunnelProvider to nest together.

Ultimately, at the bottom of any stack of transports, there is a either a PacketTunnelProvider or a TCPConnection obtained from calling createTCPConnectionThroughTunnel on a PacketTunnelProvider. This represents the actual network, and all transports, nested or single instances, act as virtual networking interfaces, sitting between the application and the actual network.

When running inside of a network extension, a PacketTunnelProvider is provided based on the Network Extension frameworks' NEPacketTunnelProvider class. When running outside of a network extension, a PacketTunnelProvider is provided based on Foundation's URLSessionStreakTask networking interface.

Interfaces for representing TCP connections and UDP sessions

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

```
public protocol TCPConnection
{
      func observeState(_ callback:
            @escaping (NWTCPConnectionState, Error?) -> Void)
      var state: NWTCPConnectionState { get }
      var isViable: Bool { get }
      var error: Error? { get }
      func cancel()
      func readLength(
            _ length: Int,
            completionHandler completion:
                  @escaping (Data?, Error?) -> Swift.Void
      )
      func readMinimumLength(
            _ minimum: Int,
            maximumLength maximum: Int,
```

```
            completionHandler completion:
                  @escaping (Data?, Error?) -> Swift.Void
      )
      func write(
            _ data: Data,
            completionHandler completion:
                  @escaping (Error?) -> Swift.Void
      )
     func writeClose()
}
```

Optionally, transports can also support applications that communicate with UDP packets. In order to do this, transports should provide a constructor function that implements the UDPSession interface either instead of or in addition to the TCPConnection.

```
public protocol UDPSession {
      var state: NWUDPSessionState { get }
      var isViable: Bool { get }
      var resolvedEndpoint: NWEndpoint? { get }
      func tryNextResolvedEndpoint()
      func setReadHandler(_ handler:
            @escaping ([Data]?, Error?) -> Void, maxDatagrams: Int)
      func writeDatagram(
            _ datagram: Data,
            completionHandler: @escaping (Error?) -> Void
      )
      func writeMultipleDatagrams(
            _ datagramArray: [Data],
            completionHandler: @escaping (Error?) -> Void
      )
      var maximumDatagramLength: Int { get }
      func cancel()
}
```

Extensions for compatibility with Apple's Network Extension API

The Network Extension API 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 NEPacketTunnelProvider. Therefore, transports cannot be used directly with code that expects a Network Extension NEPacketTunnelProvider. 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 Extension types such as NEPacketTunnelProvider to Transport API types such as

PacketTunnelProvider. Once this is done, the networking code can use either Network Extension connections or transport connections interchangeably. The Transport API accomplishes this by providing a type-compatible wrapper around the Network Extension classes. The following is an example of how this is accomplished, wrapping NEPacketTunnelProvider to be a PacketTunnelProvider.

Effect on API Compatibility

This is a new API for a new language. It therefore has no effect on applications that use the Go API and so does not, in that sense, break compatibility. However, it could also not be considered compatible with the Go API. Applications will need to implement either one API or the other. This is in accordance with the design principles specified in the architectural overview of transports, which specifies that APIs should be idiomatic to the networking code style of each language.

Effect on IPC Compatibility

The Swift API does not provide an implementation of the PT IPC protocol. If this were to be provided, it would be a separate proposal. It would need to be established that there is a use case which is in need of an PT IPC-compatible wrapper for the Swift API. Implementers that do not want to call the Swift API directly have the option of using either a Network Extension wrapper to inject transports into their application without modifying their networking code, or to use the existing IPC-compatible dispatcher for the Go implementation of transports. As this proposal simple does not interact with the IPC protocol at all, no changes are required to the IPC protocol to accommodate it.

Alternatives considered

Swift has several socket APIs. There are of course traditional BSD sockets as found on many variants of Unix. For most uses, Apple discourages use of BSD sockets and instead prefers for developers to use the URLSession classes from the Foundation framework. There are also libraries such as SwiftSocket that provide a layer of abstraction on top of the sockets provided by the Apple frameworks. None of these APIs will work from inside of a Network Extension as NEPacketTunnelProvider does not provide access to them.

There is an open question as to whether PT implementers will be using transports inside of a network extension or not. It was considered that perhaps two networking APIs would be appropriate: PacketTunnelProvider for inside of Network Extensions and URLSession for direct use in normal applications. However, the alternative socket APIs also lack any sort of composability. Providing a PT-aware alternative to URLSession would be particularly challenging as the full capabilities of URLSession include HTTP and TLS libraries. There is no

obvious way to make the existing URLSession classes PT-aware or to reuse some of their code in a PT-aware alternative. However, the proposed PacketTunnelProvider can actually be used in both a Network Extension and directly in a normal application as it includes compatibility wrappers for both NEPacketTunnelProvider and URLSession. Additionally, it is composable, with the ability to wrap multiple PacketTunnelProvider implementations together. Therefore, among the proposed alternatives it was the most versatile option.