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

Go API Dialer Refinements

Proposal: 0001
Authors: Dr. Brandon Wiley
Status: Awaiting implementation

Implementation: TBD

Introduction

This feature updates the PT Go API to conform to the modern Go API for networking, provide capabilities needed by Go application developers, and clarify the semantics of networking methods which previously had ambiguous documentation.

Motivation

The PT Go API was designed to be familiar to users of the Go networking API. However, since the PT Go API was proposed, the Go networking API has changed. In particular, Go 1.7 introduced the concept of a DialContext. This proposal seeks to modernize the PT Go API to maintain conformance with the changing Go networking API in order to make it as easy as possible for application developers working in Go to adopt the PT API in place of their existing networking code.

The proposed solution seeks to meet the following goals in improving the Go API:

- Support using a custom dialer
- Allow use of an existing connection
- Provide a mechanism for cancellation of an ongoing connection

Proposed solution

The proposed revision to the Go API consists of the following points:

- Client and server implementations are separate
- Client and server implementations are provided by constructor functions
- Client constructor functions optionally accept a Context parameter
- Transports that accept the Context parameter follow the defined explicit cancellation procedure for Go networking go by monitoring the Context's Done channel.

Additionally, the proper implementation of the following functions defined in the Go networking API in the context of a transport are defined, which were previously not explicitly defined:

- Appropriate values and meaning for the “network” parameter
- Types of addresses allowed in the “address” parameter
- The return type of dialer functions
- The possible values for LocalAddr() and RemoteAddr()
- Thread safety semantics for Read(), Write(), and Close().
- Clarification on support of setting deadlines
- Clarification of UDP support in the Go API

Design

Client and Server Implementations are Separate

The Go implementation of a transport and provide either a client implementation, a server implementation, or both in a single module. Implementation of a transport is provided by constructor functions for each transport. A module can provide either one or more client constructor functions, one or more server constructor functions, or a combination and both client constructor functions and server constructor functions.
For example, the obfs4 transport module could provide the following client constructor function:

```
obfs4.DialTransport(cert string, iatMode bool, address string) net.Conn
```

Providing only this function would mean that this module only provided an obfs4 client implementation. Additionally, the obfs4 transport module could provide the following server constructor function:

```
obfs4.Listen(cert string, iatMode bool) net.Listener
```

Providing both functions means that this module provides both a client and server implementation of the obfs4 transport protocol.

Having the option for separate client and server implementations allows application developers using transports to minimize the size of their applications by including on the client code in client-side applications.

Client Constructor Functions

In the Go PT API, clients are constructed using a client constructor function. For instance, the following:

```
obfs4.DialTransport(cert string, iatMode bool, network string, address string)
net.Conn
```

In this function, “cert” and “iatMode” are parameters specific to the obfs4 transport. The “network” and “address” parameters are from the Go networking API and are optional but common to many transports.

The “network” parameter is a string which can be any of the following values: “tcp” or “udp”. It is up to the specific transport implementation to check the network parameter and cause the function to fail if the requested network type is not available for this transport.

The “address” parameter is a string which can be either an IPv4 address, IPv6 address, or DNS name. It is up to the specific transport implementation to check the address parameter and cause the function to fail if the requested address type is not available for this transport.

The return type is net.Conn. This is an interface type defined in the Go networking API. The specific implementation is provided by the transport implementation. For instance, the client constructor function for obfs4 might return an item of type “obfs4.Obfs4Conn”, which conforms to the net.Conn protocol. The returned item can also optionally conform to the additional protocols from the Go networking library, net.TCPConn and net.UDPConn.

Dialing Context

In addition to the default client constructor function, the transport can optionally accept an additional constructor function which accepts a Go standard library context.Context. For example:

```
obfs4.DialTransport(ctx context.Context, cert string, iatMode bool, network
string, address string) net.Conn
```

Contexts should be used whenever the application wants the ability to cancel the transport connection. Support for contexts is optional and is provided by the transport if the transport provides the ability to cancel connections. There is no other way to cancel a connection without using a Context, as this is the only officially support method as of Go v1.7.

Explicit Cancellation

When a client constructor function is provided by the transport which accepts a Context argument, the transport must support explicit cancellation of the transport connection by the application. This is handled internally by the transport by reading from the Context’s Done channel. When a message is received from the Done channel, this is a signal from the application that the transport should immediately cancel the transport connection.

Server Constructor Functions

In the Go PT API, servers are constructed using a server constructor function. For instance, the following:

```
obfs4.Listen(iatMode bool, network string, address string) net.Listener
```

In this function, “iatMode” is a parameters specific to the obfs4 transport. The “network” and “address” parameters are from the Go networking API and are optional but common to many transports.

The “network” parameter is a string which can be any of the following values: “tcp” or “udp”. It is up to the specific transport implementation to check the network parameter and cause the function to fail if the requested network type is not available for this transport.

The “address” parameter is a string which can be either an IPv4 address, IPv6 address, or DNS name. It is up to the specific transport implementation to check the address parameter and cause the function to fail if the requested address type is not available for this transport.

The return type is net.Listener. This is an interface type defined in the Go networking API. The specific implementation is provided by the transport implementation. For instance, the server constructor function for obfs4 might return an item of type “obfs4.Obfs4Listener”, which conforms to the net.Listener protocol. The returned item can also optionally conform to an additional protocol from the Go networking library, net.TCPListener.

Effect on API Compatibility

This is effectively a new Go API. Transports which use the PT 2.0 Go API will need to be rewritten, as will applications that use that API. However, this is also the case for application which use normal Go networking that have been updated to use the newer version of Go (1.7 and beyond). As the Go networking APIs evolve, the Go PT API should evolve with it, as one of the goals is to have the PT API resemble the system networking API.

Effect on IPC Compatibility

This proposal has no effect on IPC compatibility as it consists solely of changes to the internal API used by the dispatcher. The Go implementation of the dispatcher will need to be modified to use this new API internally, but this will have no visible external effect.

Alternatives considered

Sticking with the existing Go API is an alternative to the proposed API modifications. The existing API has the following drawbacks which are solved by this proposal:

- Transport implementations must also provide both a client and a server, even when only the client is needed by the application.
- Transports cannot use a custom dialer provide by the application, which prevents the use of more complex networking configuration such as stacking multiple transport implementations on top of each other
- Transports cannot use a pre-existing connection, which prevents the applications from more complex protocols, such as establishing a real HTTPS tunnel using the system HTTPS implementation and then passing this connection onto the transport