Go Transport API

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.

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 Go programming language, as well as associated implementation details for using and creating pluggable transports that are utilized directly in a Go 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 which is also written in the Go 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.

Table of Contents

1. Go Transport API
    1.1. Modules
        1.1.1 Module net
        1.1.2 Transport Modules
    1.2. Implementing a Transport
        1.2.1. Dealing with Configuration Parameters
        1.2.2. Wrapping the Network Connection
    1.3. Using a Transport

2. References

Appendix A. Changelog

1. Go Transport API

The Pluggable Transport Go API provides a way to use Pluggable Transports directly from Go code. It is an alternative to using the IPC interface. The Go API may be an appropriate choice when the application and the required transport are both written in Go. When either the application or the transport are written in a different language, the IPC interface provides a language-neutral method for configuring and using transports running in a separate process.

This API specification is divided into three parts. The “Modules” section provides documentation of the types and methods provided by the Pluggable Transports Go API. The “Implementing a Transport” and “Using a Transport” sections then provide context on how the API is used.

1.1. Modules

The Pluggable Transports Go API does not define any interface types. Rather, it reuses the existing interfaces from the Go standard library’s net module.

1.1.1 Module net

These interfaces are part of the Go standard library and are provided here for reference.

```
// net.Listener implements the Listener abstract interface.
// This interface is defined in the Go standard library.
type Listener interface {
	// Accept waits for and returns the next connection to the listener.
	Accept() (net.Conn, error)

	// Close closes the listener.
	Close() error

	// Addr returns the listener's network address.
	Addr() Addr
}

// net.Conn implements the Connection abstract interface.
// This interface is defined in the Go standard library.
type Conn interface {
	// The transport-specific logic for obfuscating network traffic is
	// implemented inside the methods defined in the net.Conn interface.
	//
	// Read reads data from the transport connection. This will likely also
	// require reading data from the underlying network connection. The
	// transport-specific logic for de-obfuscating network traffic is
```

```
	// implemented here.
	Read(b []byte) (n int, err error)

	// Write writes data to the connection. This may or may not result in
	// immediate writing of data to the underlying network connection. The
	// transport-specific logic for obfuscating network traffic is
	// implemented here.
	Write(b []byte) (n int, err error)

	// Close closes the transport connection. This will usually also close
	// the underlying network connection used by the transport.
	Close() error

	// These methods are also part of the net.Conn interface. They are not
	// discussed in detail here. For more information on these methods, look
	// at the official net.Conn documentation.
	LocalAddr() Addr
	RemoteAddr() Addr
	SetDeadline(t time.Time) error
	SetReadDeadline(t time.Time) error
	SetWriteDeadline(t time.Time) error
}
```

1.1.2 Transport Modules

Transport modules MUST conform to the TransportDialer interface for client factories or the TransportListener interface for server factories. A module can provide either just a client factory, just a server factory, or both.

Here are the interfaces:

```
type interface TransportDialer
{
	func Dial() (net.Conn, error)
}

type interface TransportListener {
	func Listen() (net.Listener, error)
}
```

In order to implement these interfaces, each transport MUST provide a struct which includes a constructor function and the necessary functions to conform to one or both of these interfaces. The constructor functions for the structs take transport-specific parameters, so there is no fixed interface definition. The structs below are just examples for what these interface implementations might look like for a hypothetical transport:

```
// Create outgoing transport connection
// The Dial method implements the Client Factory abstract interface.
struct ExampleClient {
      serverAddress string
}

func NewExampleClient(serverAddress string) *ExampleClient {
      return &ExampleClient{serverAddress: serverAddress}
}

func (client *ExampleClient) Dial() (net.Conn, error) {
      return net.Dial(serverAddress)
}

// Create listener for incoming transport connection
// The Listen method implements the Server Factory abstract interface.
struct ExampleServer {
      listenAddress string
}

func NewExampleServer(listenAddress string) *ExampleServer {
      return &ExampleServer{listenAddress}
}

func (server *ExampleServer) Listen() (net.Conn, error) {
      return NewExampleListener(listenAddress), nil
}

struct ExampleListener {
      listenAddress string
}

func NewExampleListener(listenAddress string) *ExampleListener {
      return &ExampleListener{listenAddress}
}

func (listener *ExampleListener) Listen() (net.Conn, error) {
      return nil, errors.New("Listen not yet implemented")
}
```

Unlike the Dial and Listen functions supplied by Go's net.Conn package, the transport Dial and Listen functions do not take a parameter for a server address as this information is included in the transport configuration options. For consistency, transports SHOULD name this option "serverAddress" and it SHOULD have a format of <address>:<port>. Unless otherwise specified in the documentation of the specific transport being used, the address can be an IPv4 IP address, an IPv6 IP address, or a domain name. Not all transports require a server address and

some will require multiple server addresses, so this convention only applies to the case where the transport requires a single server address

Transport constructor functions MAY take as a parameter a network parameter, which is a string. The only valid values for the network parameter are “tcp” or “udp”. If the transport constructor function takes a network parameter, the transport implementation MUST check the parameter to determine if it is compatible with the specific transport and return an error in the case of incompatibility. Specifying “tcp” or “udp” indicates whether the returned net.Conn object should implement TCP or UDP semantics when sending and receiving data. It does not specify whether the transport should actually use TCP or UDP when communicating over the network. A transport MAY provide only TCP semantics, only UDP semantics, or both.

The Dial and Listen functions return a tuple of an optional value and an optional error. Either the value or the error should be set, while the other is nil. It is invalid to return both a value and an error. It is also invalid to return neither a value or an error.

Here are some examples of valid return combinations:

```
func (client *ExampleClient) Dial() (net.Conn, error) {
      return net.Dial(network, “127.0.0.1:1234”, nil
}

func (client *ExampleClient) Dial() (net.Conn, error) {
      return nil, errors.New(“Error”)
}
```

Here are some example of invalid return combinations:

```
func (client *ExampleClient) Dial() (net.Conn, error) {
      return net.Dial(network, “127.0.0.1:1234”, errors.New(“Error”)
}

func (client *ExampleClient) Dial() (net.Conn, error) {
      return nil, nil
}
```

An example of a constructor function that takes a network parameter:

```
struct ExampleClient {
      serverAddress string
      network string
}

func NewExampleClient(serverAddress string, network string) *ExampleClient {
      return &ExampleClient{serverAddress: serverAddress}
}
```

```
func (client *ExampleClient) Dial() (net.Conn, error) {
      if network == "tcp" {
            return ExampleTCPConn(serverAddress), nil
      } else if network == "udp" {
            return ExampleUDPConn(serverAddress), nil
      } else {
            return nil, errors.New("Unknown network string")
      }
}
```

Transport constructor functions MAY take as a parameter a custom dialer function to use when making network connections. For example:

```
struct ExampleClient {
      serverAddress string
      dialer func() (net.Conn, error)
}

func NewExampleClient(serverAddress string, dialer func() (net.Conn, error))
*ExampleClient {
      return &ExampleClient{serverAddress, dialer}
}

func (client *ExampleClient) Dial() (net.Conn, error) {
      return dialer()
}
```

Transport constructor functions can optionally take as a parameter a dialer context to use when making network connections. The primary reason for taking a context is to allow for cancellation when dialing a network connection. Therefore, transports that provide a constructor function that takes a dialer context MUST follow the defined explicit cancellation procedure for Go networking by monitoring the Context's Done channel.

Here is an example of a constructor function that takes a custom dialer context:

```
import "context"

struct ExampleClient {
      serverAddress string
      context Context
}

func NewExampleClient(serverAddress string, context Context) *ExampleClient {
      return &ExampleClient{serverAddress, context}
}
```

```
func (client *ExampleClient) Dial() (net.Conn, error) {
      return net.DialContext(context, “tcp”, serverAddress)
}
```

1.1.3 Logging

For each language-specific API, there is a language-specific logging mechanism, with standardized log levels used across all languages. For the Go API, as the built-in Go standard library logging system does not support the required log levels, the open source logging package “golog” is used. The full documentation for golog can be found here: https://github.com/kataras/golog The golog library uses globally available functions for configuration and logging, so no changes are needed to the API used to initialize transport modules.

1.1.2.1 Using Go Logging from Inside a Transport

Import the logging library

```
import "github.com/kataras/golog"
```

The log has functions for each loglevel. The permitted logging functions are Debug, Info, Warning, and Error.

Examples of logging:

```
golog.Debug("debug”)
golog.Info("info")
golog.Warn("warn")
golog.Error("err")
```

1.1.2.2 Setting up a Log in a Go Application

Applications must globally configure the log so that it can be used by the transport. The log can be configured with application-specific options.

Import the logging library

```
import "github.com/kataras/golog"
```

Set the logging output. This sets all logging calls globally to log to stderr.

```
golog.SetOutput(os.Stderr)
```

Set the loglevel globally for all logging calls. This sets the loglevel to DEBUG.

```
golog.SetLevel("debug")
```

1.2. Implementing a Transport

In order to implement a transport, at least one transport interface must be implemented. For a client, a TransportDialer implementation is required and for a server a TransportListener implementation is required. A transport may implement both the client and server. The transport constructor function, being a normal Go 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.

Listening for incoming connections is handled by an instance of the net.Listener interface return by the TransportListener's Listen() function. On the net.Listener, an Accept() function allows for accepting a new incoming transport connection and the Close() function stops listening for incoming connections.

The transport will also need to implement instances of the net.Conn interface. The Dial and Listen functions both return instances of the net.Conn interface. In most cases, these will be different implementations of the interface, one for encoding traffic into the transport's specific protocol and the other for decoding this traffic.

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

Transports are required to implement the net.Conn interface, which includes some methods related to setting deadlines, such as SetDeadline(). For type conformance in Go, these methods must exist on the transport implementation. However, transports are not required to fully implement them and can instead return an error. Transports seeking to implement these methods fully should seek guidance from the Go standard library net.Conn documentation.

Implementing the net.Conn interface also requires providing the LocalAddr() and RemoteAddr() methods, which return, respectively, the local and remote IP addresses of the connection. These values will not always make sense in the context of a transport. Specific values for these functions are not defined in this specification and applications using transports should not rely on these functions. Transports MAY use arbitrary default values for these functions or delegate to the underlying network connection, if one exists.

The Go standard library does not specify thread safety requirements for net.Conn functions such as Read() and Write(). Therefore, this specification also does not define specific thread

safety requirements. Applications using transports or using Go networking directly should not assume that methods are thread-safe.

1.2.1. 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 at least one constructor function and the type signature for that function specifies the required parameters. For instance, here is an example client constructor for obfs4:

*func NewObfs4Client(nodeID *ntor.NodeID, publicKey *ntor.PublicKey, sessionKey *ntor.Keypair, iatMode int, serverAddress string) *Obfs4Dialer*

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.

1.2.2. Wrapping the Network Connection

The transformations provided by each transport to turn data into traffic and back again are provided by the net.Conn implementations returned by the Dial and Accept functions. The transport net.Conn possibly wraps other net.Conn instances representing the network connection. A call to the transport net.Conn.Write() would then be translated into one or more calls to the network net.Conn.Write(). Similarly, a call to the transport net.Conn.Read() would be translated into one or more calls to the network net.Conn.Read(). While this is a common strategy for implementing a transport, the transport is free to use any means to get the data across the network and may or may not wrap a net.Conn as its means of communicating with the network.

1.3. 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.

2. 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

Appendix A. Changelog

PT 3.0

- Implemented proposal 0005 - Improved Error Handling in Go API
- Implemented proposal 0011 - Improve Logging in APIs
- Implemented proposal 0013 - Migrate Go API from Factory Functions to a Factory Interface
- Implemented proposal 0014 - Make Server Address Part of Client Options

PT 2.1

- Implemented proposal 0001 - Go API Dialer Refinements
    - Client and server implementations are separate
    - 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
    - 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.
    - 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
- Implemented proposal 0002 - Modularization of Specification

PT 2.0, Draft 3

- Removed TransportConn and TransportListener in favor of net.Conn and net.Listener

PT 2.0, Draft 2

- Modified Go examples to use correct Go syntax
- Renamed pt module in Go examples to base to avoid naming conflict with goptlib
- Clarified Go examples with more details on how to implement a transport in Go
- Removed SSH transport example
- Standardized use of Transports API and Dispatcher IPC language throughout