Migrate Go API from Factory Functions to a Factory Interface

Proposal: 0013
Authors: Dr. Brandon Wiley, Operator Foundation
Status: Initial proposal

Implementation: TBD

Introduction

The current revision of the Go API specifies that implementations of the Go API should provide factory functions for clients and servers. Under this proposal, these isolated functions would be replaced with interfaces containing these functions.

Motivation

There are two motivations for this change. The first is to reach closer parity with the other APIs, which use an approach more similar to this proposal than what is in the current revision of the specification. The second is recent work that has been done on the Optimizer transport, which is a transport that wraps other transports. Having transports represented as bare separate functions makes it difficult to provide this sort of wrapping. In order to facilitate further research into transports that wrap other transports, packing the factory functions as interfaces is proposed.

Proposed solution

The Dial and Listen functions described in the Go API specification will be replaced with interfaces that contain these functions.

Design

The new interfaces are defined as follows:

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

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

The factory functions that take transport-specific options are now replaced with a factory function that returns instances of these interfaces.

For example:

```
struct ExampleTransport {
     config string
}

func New(config string) *ExampleTransport {
     return &ExampleTransport{config: config}
}

func (transport *ExampleTransport) Dial() (net.Conn, error) {
     // implementation ellided
}

func (transport *ExampleTransport) Listen() (net.Listener, error) {
     // implementation ellided
}
```

In this example, a single struct type is used to conform to both interfaces. However, it is also possible to have separate structs for client and server or to only provide one or the other.

Here is an example of usage:

```
var dialer TransportDialer
dialer = New("config")
conn := dialer.Dial()

var transportListener TransportListener
transportListener = New("config")
listener := transportListener.Listen()
```

An advantage of this approach is that it makes it easier to manage collections of transports. For instance, it is simple to put several transports into an array.

Here is an example:

```
var transports []TransportDialer
transports = []TransportDialer{New(“config 1”), New(“config 2”)}
```

This functionality makes it easier to write transports that wrap other transports, as the transports can be passed around as interface types rather than as function types. This sort of single-function interface has a precedent in the Go standard library “net” package, in which the Dialer interface is defined to have a single Dial function. In this way, this proposal mimics the approach found in the Go standard library. Of particular interest is that this array can include instances of different transports as long as they conform to the TransportDialer interface.

Effect on API Compatibility

This change is not backwards-compatible and would require a new major version number.

Effect on IPC Compatibility

This proposal only affects the Go API.

Alternatives considered

The alternative to having interface types is to pass around transports as bare functions. This complicates things such as storing them in arrays, maps, and structs.

For instance, here is how the above example of having an array of transport client would look using the current Go API:

```
package ExampleTransport

func Dial(config string) (net.Conn, error)
{
      // implementation ellided
}

...

import “ExampleTransport”

var transports [](string,func(string)net.Conn)
transports = [](string,func(string)net.Conn){
      (“config 1”,ExampleTransport.Dial)
      (“config 2”,ExampleTransport.Dial)
}
```

In addition to being more complex and harder to read, this construction also does not allow arrays containing multiple different types of transports. As each Dial function has a different signature, the type of the factory function varies. By switching to an interface, all of this complexity is hidden and instances of different transports can be used interchangeably. This provides a new level of “pluggability” for code which manages multiple different transports.