Go client-side interceptors: thoughts

zellyn@squareup.com - 2016-03-23

Retries

The most complicated thing we wish to do with client-side interception is parallel retries on idempotent requests. We mark RPC idempotency with proto options, and when an idempotent Call reaches its “retry timeout”, but still has time remaining before the context timeout, we fire off a new request in parallel, hoping that the original request will return first.

When to pick/load-balance

The picker call cannot happen before the interceptors are run. The Java client-side interceptors do the right thing: the picker/load-balancer happens inside the handler given to interceptors. So an interceptor can call a handler multiple times.

CallOptions

It’s unclear when/where the CallOptions should be run, if multiple requests are made. The existing ones (which capture Header and Trailer information) should probably only run on the successful request.

Curious to see what happens with Streaming calls, since they have similar multiple-message problems, I notice that CallOptions are simply unimplemented for Streaming calls :-)

Output parameters

In the current design, `grpc.Invoke` takes both input and output `interface{}` parameters, instead of an output return value, so that the type is already set. If multiple calls are made for retries, it makes things tricky: all calls cannot write to the same output parameter. Some kind of output factory is needed, or the proto codec needs to be called only for the one successful call. (or coordinate writes so that happen sequentially, and the successful one happens last.)

One approach

One approach Josh Humphries (jh@squareup.com) and I discussed for a minimally invasive change was adding a `grpc.Channel` interface, and using it in the generated protobuf grpc code. The generated code only calls two methods on `ClientConn`:

```
type Channel interface {
```

```
  Invoke(ctx context.Context, method string, args interface{}, replyFactory func()
interface{}, opts ...CallOption) (error)
  NewClientStream(ctx context.Context, desc *StreamDesc, method string, opts ...CallOption)
(ClientStream, error)
}
```

clientConn would need two new methods that just call `grpc.Invoke` and `grpc.NewClientStream`.

```
func (c *clientConn) Invoke(ctx context.Context, method string, args interface{},
replyFactory func() interface{}, opts ...CallOption) (error) {
  return grpc.Invoke(ctx, method, args, replyFactory(), c, opts…)
}

func (c *clientConn) NewClientStream(ctx context.Context, desc *StreamDesc, method string,
opts ...CallOption) (ClientStream, error) {
  Return grpc.NewClientStream(ctx, desc, method, c, opts…)
}
```

The generated code changes are straightforward:

```
// NewRouteGuideClient takes a Channel instead of a ClientConn.
func NewRouteGuideClient(ch grpc.Channel) RouteGuideClient {
	return &routeGuideClient{ch}
}

// featureFactory generates new Feature proto messages.
func featureFactory() *Feature {
	return new(Feature)
}

func (c *routeGuideClient) GetFeature(ctx context.Context, in *Point, opts
...grpc.CallOption) (*Feature, error) {
	out, err := c.ch.Invoke(ctx, "/routeguide.RouteGuide/GetFeature", in, featureFactory,
opts...)
	if err != nil {
		return nil, err
	}
	return out.(*Feature), nil
}

func (c *routeGuideClient) RecordRoute(ctx context.Context, opts ...grpc.CallOption)
(RouteGuide_RecordRouteClient, error) {
	stream, err := c.ch.NewClientStream(ctx, &_RouteGuide_serviceDesc.Streams[1],
"/routeguide.RouteGuide/RecordRoute", opts...)
	if err != nil {
		return nil, err
	}
	x := &routeGuideRecordRouteClient{stream}
	return x, nil
}
```

Note: This approach doesn't solve the problem with `callOptions` being called multiple times. Since they run with `defer`, they'll be called even if the later Calls are cancelled.