gRPC

The load balancing gotchas and the connection pool solution

By: Ruba Zeibak (Stitch)

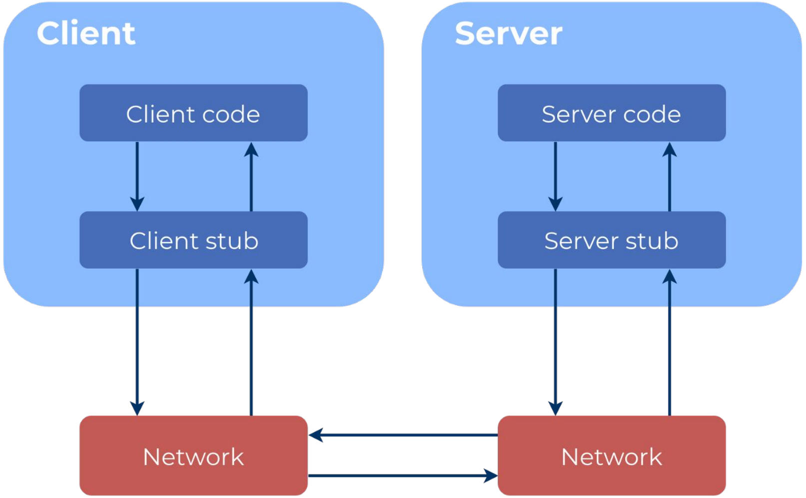
Agenda

- What is gRPC? A short introduction
- HTTP/2 - The secret weapon of gRPC
- gRPC at Stitch
- The FLOW_CONTROL error
- gRPC in k8s and load balancing
- Demo - it’s GO time!

RPC Architecture

Remote Procedure Call

RPC protocols allow to invoke a function/method on a remote server and get the result in the same format regardless of where it is executed



What is gRPC?

- ➔ gRPC is an open-source Remote Procedure Call (RPC) framework that is used for high-performance communication between services.

- ➔ It is an efficient way to connect services written in different languages with pluggable support for load balancing, tracing, health checking, and authentication

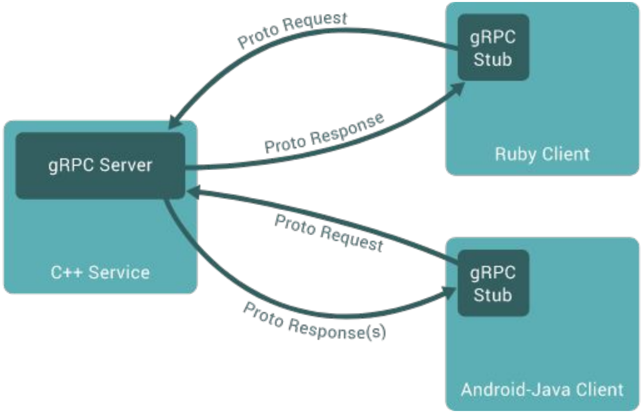
Is g for Google?

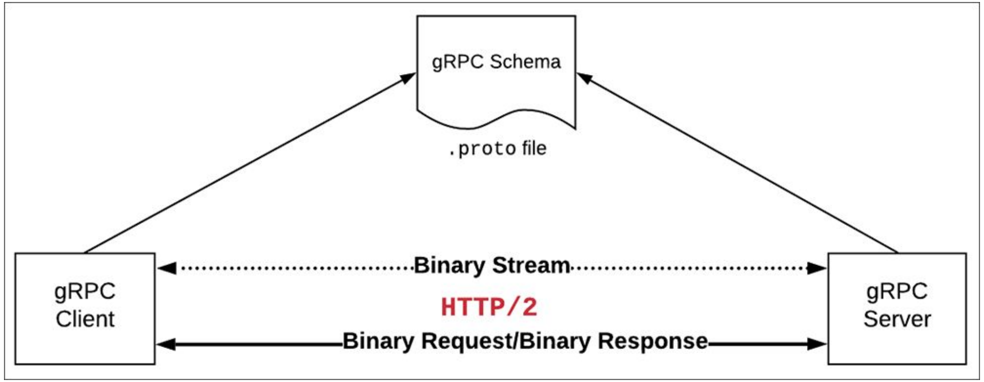
[GRPC Core: g_stands_for](#)
['g' stands for something different every gRPC release:](#)
[grpc.github.io](#)

- 1.0 'g' stands for 'gRPC'
- 1.1 'g' stands for 'good'
- 1.2 'g' stands for 'green'
- 1.3 'g' stands for 'gentle'
- 1.4 'g' stands for 'gregarious'
- 1.6 'g' stands for 'garcia'
- 1.7 'g' stands for 'gambit'
- 1.8 'g' stands for 'generous'
- 1.9 'g' stands for 'glossy'
- 1.10 'g' stands for 'glamorous'
- 1.11 'g' stands for 'gorgeous'
- 1.12 'g' stands for 'glorious'
- 1.13 'g' stands for 'gloriosa'
- 1.14 'g' stands for 'gladiolus'
- 1.15 'g' stands for 'glider'
- 1.16 'g' stands for 'gao'
- 1.17 'g' stands for 'gizmo'
- 1.18 'g' stands for 'goose'
- 1.19 'g' stands for 'gold'
- 1.20 'g' stands for 'godric'
- 1.21 'g' stands for 'gandalf'
- 1.22 'g' stands for 'gale'
- 1.23 'g' stands for 'gangnam'
- 1.24 'g' stands for 'ganges'
- 1.25 'g' stands for 'game'
- 1.26 'g' stands for 'gon'
- 1.27 'g' stands for 'guantao'
- 1.28 'g' stands for 'galactic'
- 1.29 'g' stands for 'gringotts'
- 1.30 'g' stands for 'gradius'
- 1.31 'g' stands for 'galore'
- 1.32 'g' stands for 'giggle'
- 1.33 'g' stands for 'geeky'
- 1.34 'g' stands for 'gauntlet'
- 1.35 'g' stands for 'gecko'
- 1.36 'g' stands for 'gummybear'
- 1.37 'g' stands for 'gilded'
- 1.38 'g' stands for 'guadalupe_river_park_conservancy'
- 1.39 'g' stands for 'goofy'
- 1.40 'g' stands for 'guileless'

Protocol Buffers

gRPC uses protocol buffers as the interface definition language for serialization and communication instead of JSON/XML.

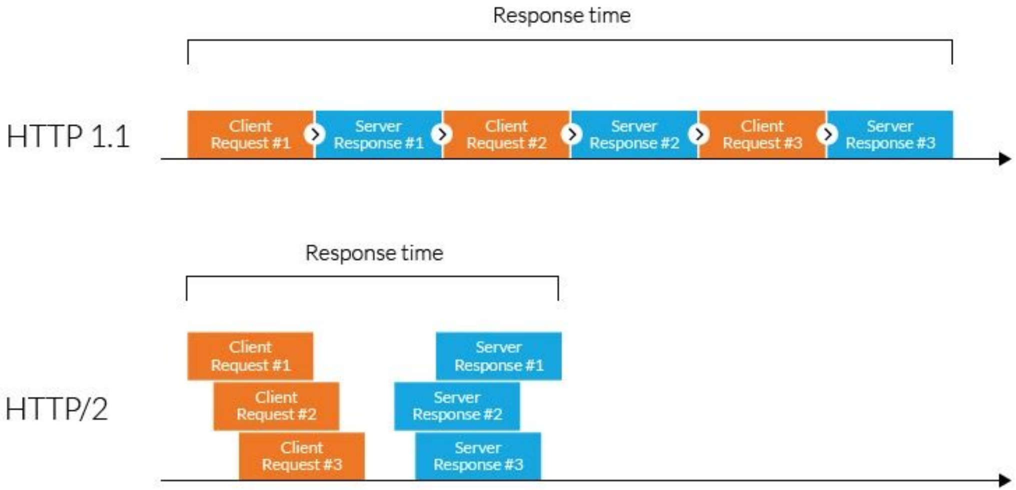




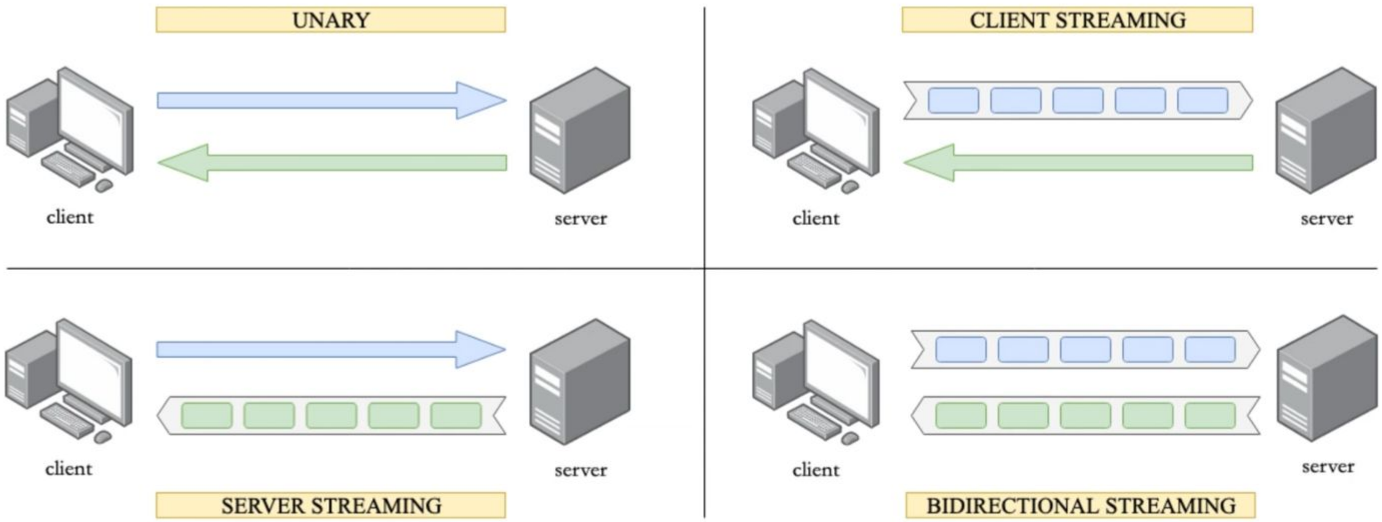
HTTP/2 - The secret weapon of gRPC

HTTP/1.1 vs. HTTP/2 Protocol

- Binary protocols
- Multiplexing
- Header compression
- Server push
- Increased security



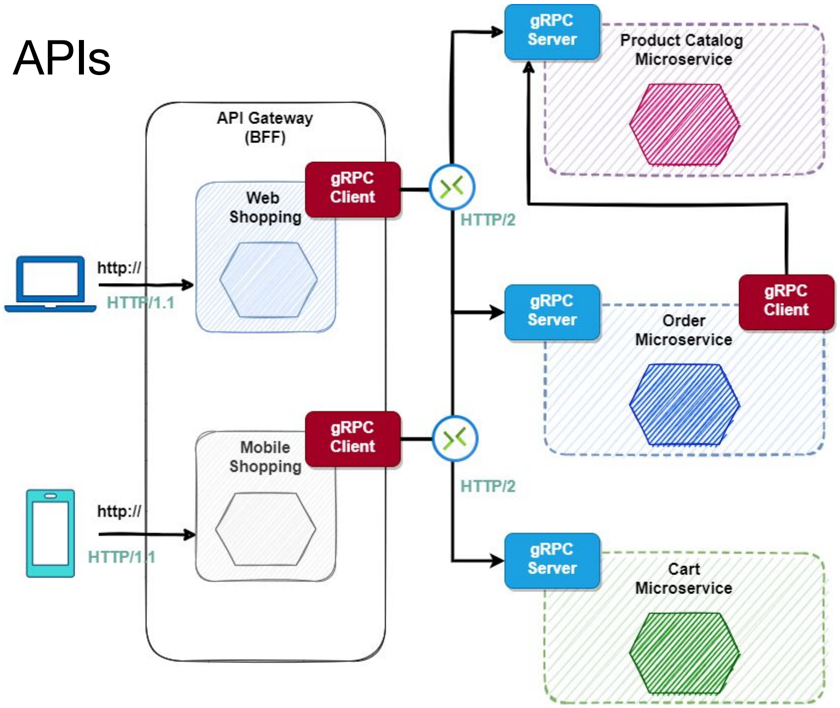
Types of gRPC



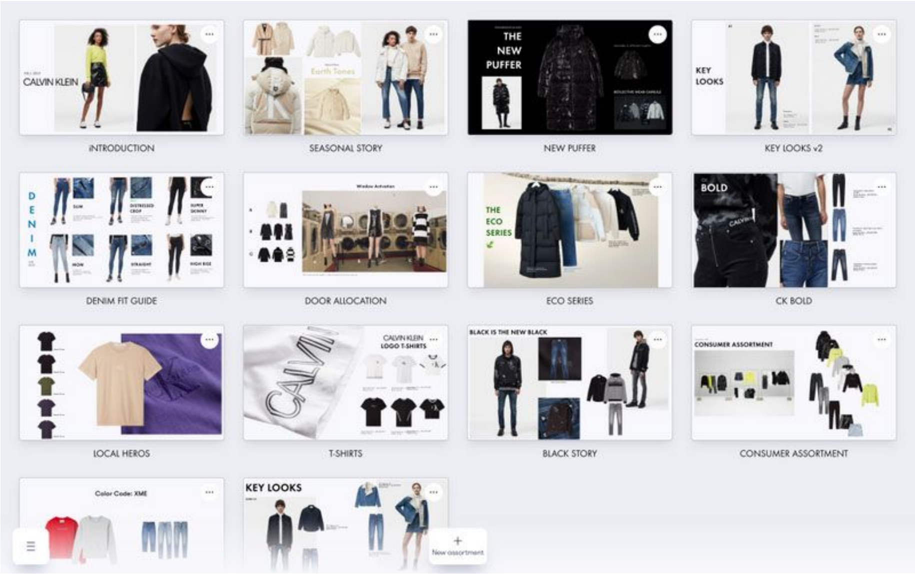
What is gRPC good for?

- Faster compared to JSON based RESTful APIs
- Strong type and well-defined interface
- Polyglot
- Stream support
- Built in support for auth, load balancing,

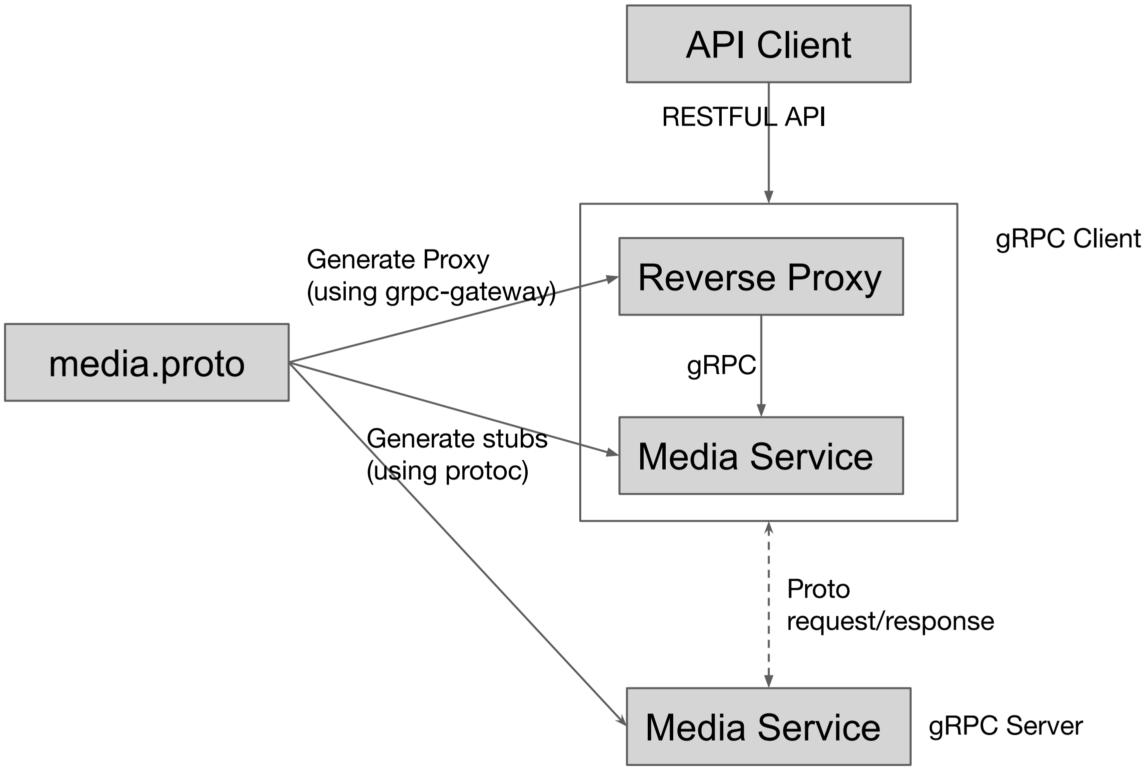
encryption, compression and error handling



The Digital Showroom at Stitch



gRPC at Stitch



HTTP/2 Flow Control

The unexpected error

Flow Control allows a receiver to stop a sender from sending it data if it isn't yet ready to process, perhaps because it's too busy to process any more incoming data.

level=error msg="finished unary call with code Internal" api=grpc error="rpc error: code = Internal desc = create the media: stream error: stream ID 7977; FLOW_CONTROL_ERROR; received from peer" grpc.code=Internal

gRPC and k8s

isn’t that straight forward after all

https://kubernetes.io/blog/2018/11/07/grpc-load-balancing-on-kubernetes-without-tears/

gRPC Load Balancing on Kubernetes without Tears

Wednesday, November 07, 2018

Author: William Morgan (Buoyant)

Load balancing gRPC

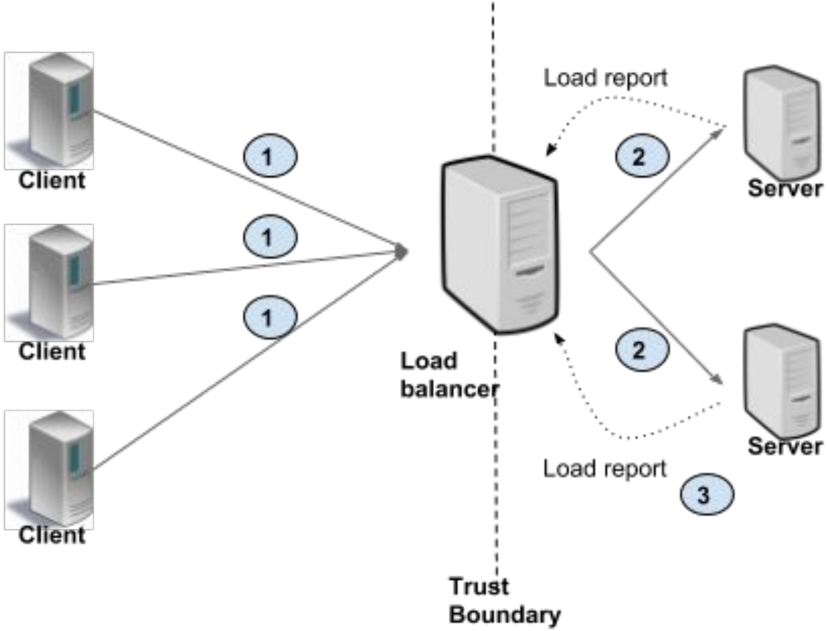
In k8s

In order to do gRPC load balancing, we need to shift from connection balancing (L3/L4 which is the k8s default) to *request* balancing (L5/L7).

In other words, we need to open an HTTP/2 connection to each destination, and balance *requests* across these connections.

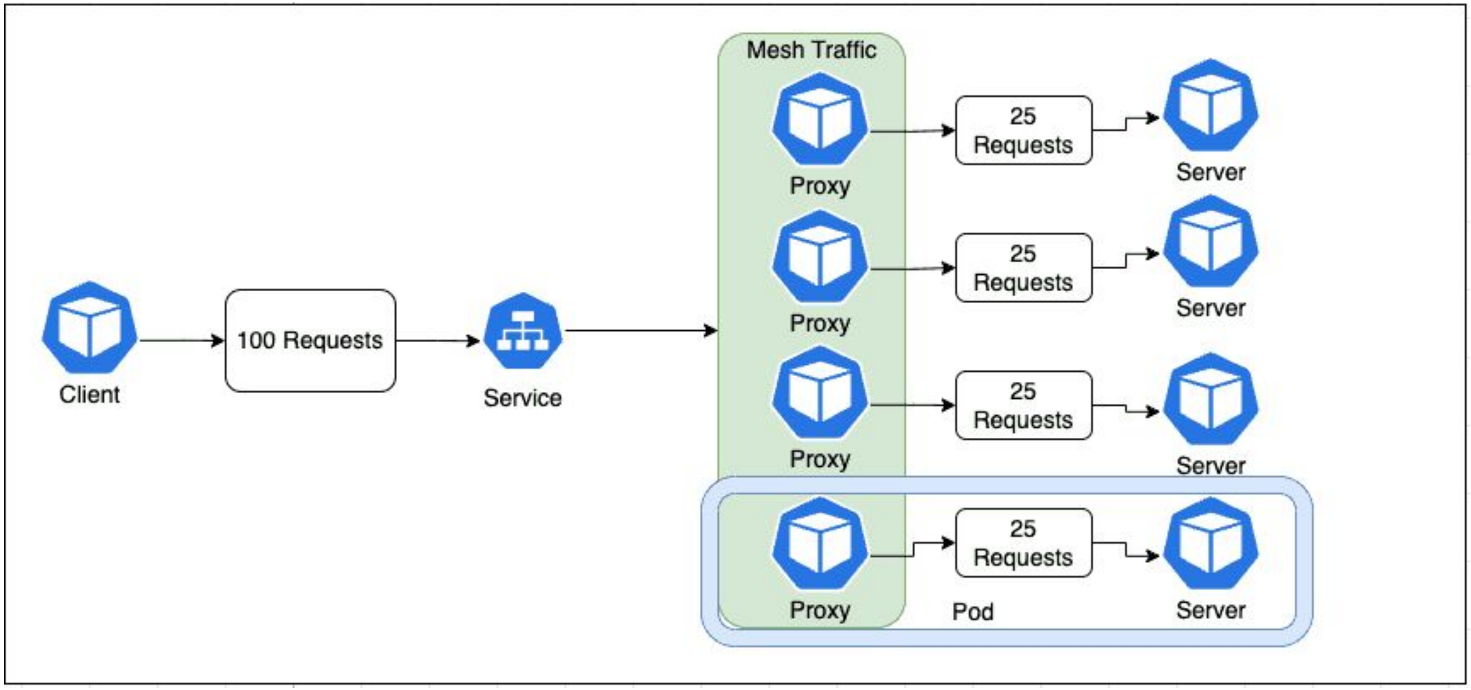
Proxy Load balancing

Also known as Server-Side LB



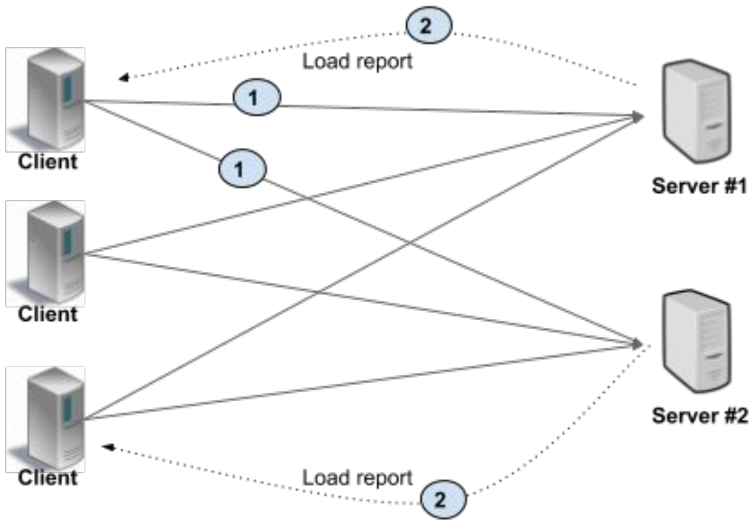
Proxy LB in K8s

Using service mesh like Linkerd or istio

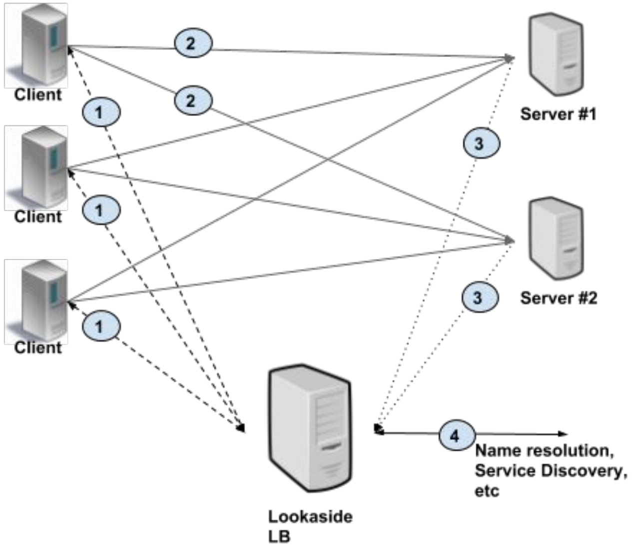


Client Side Load balancing

Thick LB



Lookaside LB



Client side LB in K8s

Client-side load balancing using headless service

You can do client-side **round-robin** load-balancing using **Kubernetes headless service.** This simple load balancing works out of the box with gRPC.

The downside is that it does not take into account the load on the server.

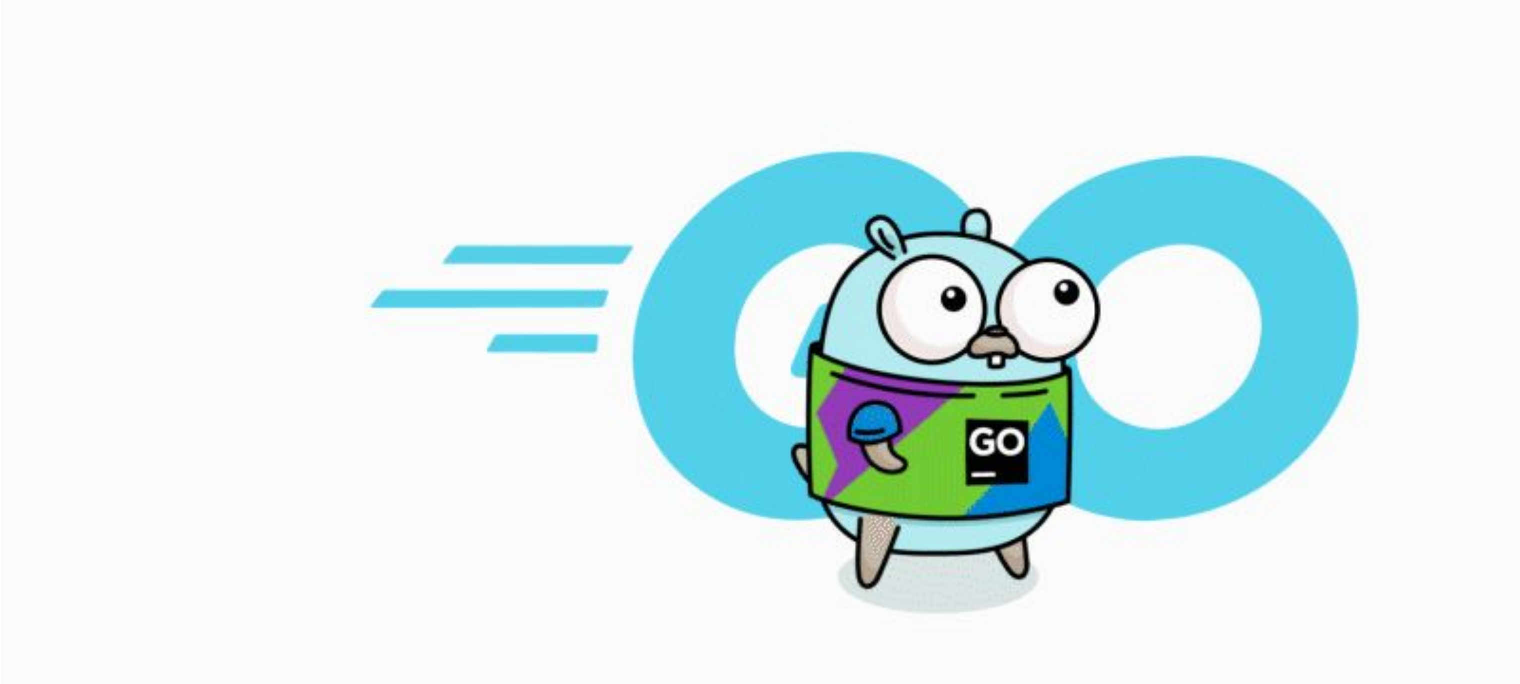
The gRPC connection pool solution

The chosen solution: A gRPC connection pool on client side combined with a server side TCP (Layer4) load balancer.

This will create a pool of client connections initially, and re-use this pool of connections for subsequent gRPC requests.

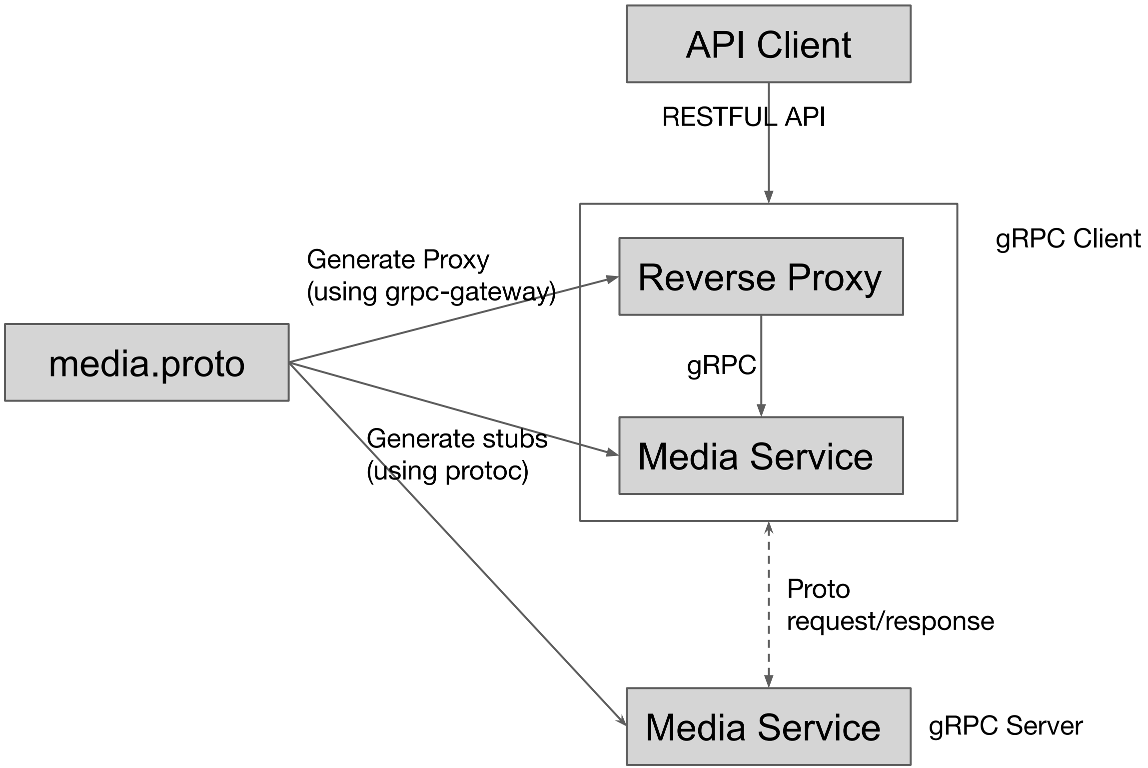
Using https://github.com/processout/grpc-go-pool

It’s finally Go time



```
syntax = "proto3";

package media;

message DownloadRequest {
    string    url = 1;
}

message MediaDimensions {
    //height of the image
    double height = 1;
    //width of the image
    double width = 2;
}

service Medias {
    //Download
    //
    // Download Media example for Go meetup Amsterdam May 2022
    rpc Download (DownloadRequest) returns (MediaDimensions);
}
```

gRPC at Stitch



```go
// RegisterMediasHandlerClient registers the http handlers for service Medias
// to "mux". The handlers forward requests to the grpc endpoint over the given implementation of "MediasClient".
// Note: the gRPC framework executes interceptors within the gRPC handler. If the passed in "MediasClient"
// doesn't go through the normal gRPC flow (creating a gRPC client etc.) then it will be up to the passed in
// "MediasClient" to call the correct interceptors.
func RegisterMediasHandlerClient(ctx context.Context, mux *runtime.ServeMux, client MediasClient) error {
    mux.Handle( meth: "POST", pattern_Medias_Download_0, func(w http.ResponseWriter, req *http.Request,
     pathParams map[string]string) {
        ctx, cancel := context.WithCancel(req.Context())
        defer cancel()
        inboundMarshaler, outboundMarshaler := runtime.MarshalerForRequest(mux, req)
        rctx, err := runtime.AnnotateContext(ctx, mux, req,  rpcMethodName: "/media.Medias/Download")
        if err != nil {
            runtime.HTTPError(ctx, mux, outboundMarshaler, w, req, err)
            return
        }
        resp, md, err := request_Medias_Download_0(rctx, inboundMarshaler, client, req, pathParams)
        ctx = runtime.NewServerMetadataContext(ctx, md)
        if err != nil {
            runtime.HTTPError(ctx, mux, outboundMarshaler, w, req, err)
            return
        }

        forward_Medias_Download_0(ctx, mux, outboundMarshaler, w, req, resp, mux.GetForwardResponseOptions()...)

    })

    return nil
}
```

```go
// MediasClient is the client API for Medias service.
//
// For semantics around ctx use and closing/ending streaming RPCs, please refer to https://pkg.go.dev/google.golang.
 org/grpc/?tab=doc#ClientConn.NewStream.
type MediasClient interface {
	//Download
	//
	// Download Media example for Go meetup Amsterdam May 2022
	Download(ctx context.Context, in *DownloadRequest, opts ...grpc.CallOption) (*MediaDimensions, error)
}

type mediasClient struct {
	cc grpc.ClientConnInterface
}

func NewMediasClient(cc grpc.ClientConnInterface) MediasClient {
	return &mediasClient{ cc: cc}
}

func (c *mediasClient) Download(ctx context.Context, in *DownloadRequest, opts ...grpc.CallOption) (*MediaDimensions, error) {
	out := new(MediaDimensions)
	err := c.cc.Invoke(ctx,  method: "/media.Medias/Download", in, out, opts...)
	if err != nil : nil, err
	return out, nil
}
```

```go
func NewMediasClient(cc grpc.ClientConnInterface) MediasClient {
	return &mediasClient{cc: cc}
}
```

```go
// ClientConnInterface defines the functions clients need to perform unary and
// streaming RPCs.  It is implemented by *ClientConn, and is only intended to
// be referenced by generated code.
type ClientConnInterface interface {
	// Invoke performs a unary RPC and returns after the response is received
	// into reply.
	Invoke(ctx context.Context, method string, args interface{}, reply interface{}, opts ...CallOption) error
	// NewStream begins a streaming RPC.
	NewStream(ctx context.Context, desc *StreamDesc, method string, opts ...CallOption) (ClientStream, error)
}
```

```go
//implements grpc.ClientConnInterface
type connClientWithPool struct {
    endpoint string
    pool     *grpcpool.Pool
}

// Invoke performs a unary RPC and returns after the response is received
// into reply.
func (cp connClientWithPool) Invoke(ctx context.Context, method string, args interface{}, reply interface{},
 opts ...grpc.CallOption) error {
    conn, err := cp.pool.Get(ctx)
    if err != nil {
        err = errors.Wrapf(err,  format: "get pool connection method %v, on endpoint %v on invoke", method, cp.endpoint)
        return err
    }

    defer func() {
        err := conn.Close()
        if err != nil {
            logr.Warn(errors.Wrap(err,  message: "return connection back to pool"))
        }
    }()

    logr.Debug( args...: "invoking the method")
    if err := conn.Invoke(ctx, method, args, reply, opts...); err != nil {
        err = errors.Wrapf(err,  format: "invoke method %v, on endpoint %v", method, cp.endpoint)
        logr.Warn(err)
        return err
    }
    return nil
}
```

```go
func main() {
	ctx := context.Background()

	cfg := config{}
	if err := envconfig.Process( prefix: "", &cfg); err != nil {
		log.Fatal(errors.Wrap(err,  message: "get env configs"))
	}
	mux := runtime.NewServeMux()
	opts := []grpc.DialOption{
		grpc.WithConnectParams(grpc.ConnectParams{
			Backoff: backoff.DefaultConfig,
		}),
	}
	if err := initMedia(ctx, mux, cfg, opts); err != nil {
		log.Fatal(errors.Wrap(err,  message: "init endpoints"))
	}
	server := pkghttp.NewServer(logr, mux,  basepath: "/", cfg.RESTAPIHost, cfg.RESTAPIPort)
	if err := server.Close(); err != nil {
		logr.Error(err)
	}
}
```

```go
func initMedia(ctx context.Context, mux *runtime.ServeMux, cfg config, opts []grpc.DialOption) error {
    poolCfg := poolConfig{
        max:              cfg.GRPCPoolMax,
        init:             cfg.GRPCPoolInit,
        timeoutInSeconds: cfg.GRPCPoolIdleTimeoutSeconds,
    }
    endpoint := net.JoinHostPort(cfg.MediaHost, cfg.MediaPort)
    conn, err := grpcDialWithPool(endpoint, poolCfg, opts)
    if err != nil {
        return errors.Wrap(err, message: "connect to media")
    }
    // Methods below always return nil error
    _ = media.RegisterMediasHandlerClient(ctx, mux, media.NewMediasClient(conn))
    return nil
}
```

```go
// Return a new grpc client connection that implements grpc.ClientConnInterface  with connection pool that returns the
// first available client connection.
func grpcDialWithPool(endpoint string, poolCfg poolConfig, opts []grpc.DialOption) (conn *connClientWithPool, err error) {
	pool, ok := pools[endpoint]
	if !ok {
		pool, err = grpcpool.New(factory(endpoint, opts), poolCfg.init, poolCfg.max, poolCfg.timeoutInSeconds)
		if nil != err {
			logr.Errorln(err)
			return conn: nil, err
		}
		pools[endpoint] = pool
	}
	logr.Infof( format: "grpcpool endpoint:%s, capacity:%d,available:%d", endpoint, pool.Capacity(), pool.Available())

	return &connClientWithPool{
		endpoint: endpoint,
		pool:     pool,
	}, err
}
```

```go
func factory(endpoint string, opts []grpc.DialOption) func() (*grpc.ClientConn, error) {
    return func() (*grpc.ClientConn, error) {
        ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
        defer cancel()
        conn, err := grpc.DialContext(ctx, endpoint, opts...)
        if err != nil {
            return nil, err
        }
        return conn, nil
    }
}
```

Was this a good choice?

Advantages:

- Reusing of existing L4 K8s LB - and hence all pods share the load
- Still uses Multiplexing since the pool.Get() returns the next available client which reuses all the connections in the pool
- No more FLOW_CONTROL errors

Disadvantages:

- Thickening the client logic
- Not as scalable as a proper LB solution

Questions?