StackRox

Talking to Go gRPC Services via HTTP/1

gRPC Conf 2020 - July 27, 2020

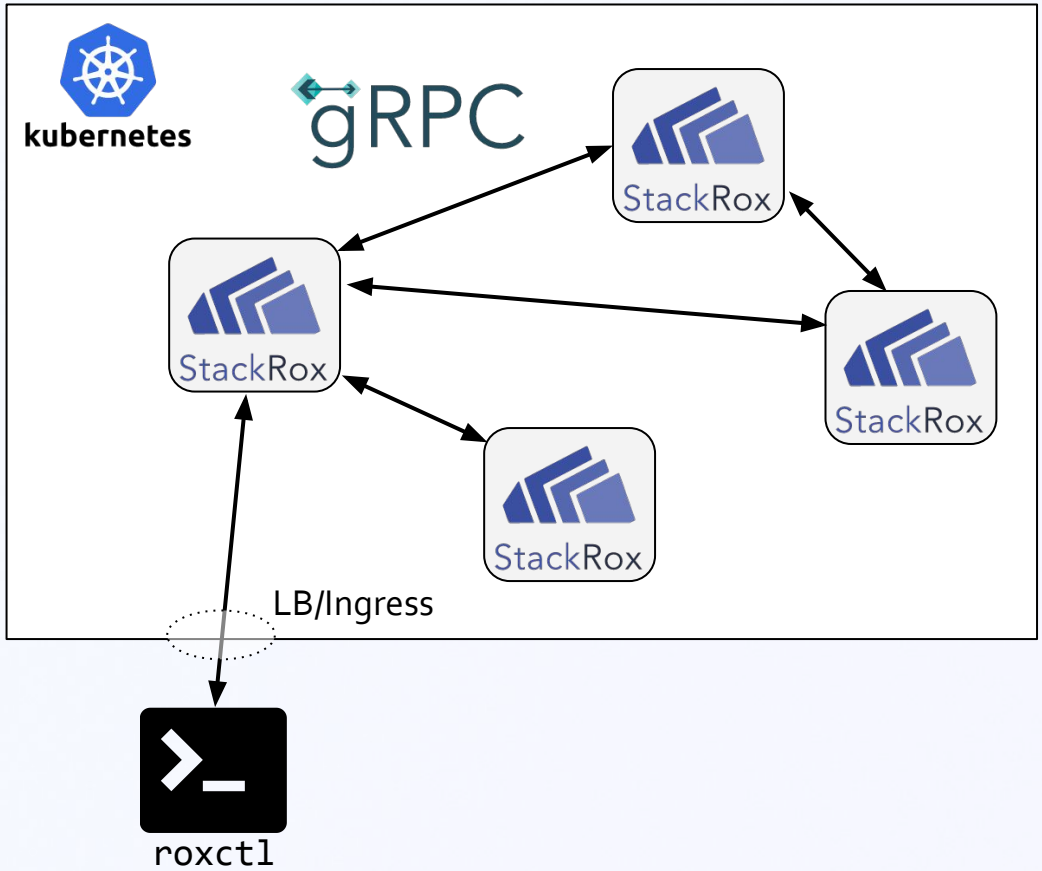
Malte Isberner
Managing Director, StackRox GmbH, Bochum, Germany
Principal Engineer, StackRox Inc., Mountain View, USA

Overview

1. Introduction & Problem Statement
2. Why gRPC Requires HTTP/2
3. gRPC-Web and Downgrading
4. Go Library: Implementation & Usage Example
5. Conclusion

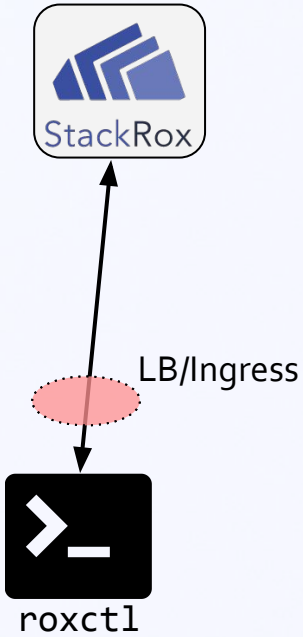
Problem Statement

- Fleet of microservices deployed in a K8s cluster
- gRPC for (intra-cluster) service-to-service communication 😀
- gRPC for CLI-to-service communication through LB 😱



The Problem with Load Balancers

- gRPC is based on HTTP/2
- Many LBs don't like (or support) HTTP/2
  - More precisely: **L7** LBs
  - Most prominent example: AWS ALB
- Viable* options:
  - Use TCP LB only (e.g., *TLS-passthrough*)
  - Use an LB that supports end-to-end HTTP/2
    - A.o. GCP HTTP2 LB, nginx 1.13.10+, HAproxy 1.9.2+
  - (Use Envoy with gRPC HTTP/1.1 bridge)
- * technical viability ≠ practical viability:
  - Can't tell customers not to use ALB but TLS passthrough w/o managed certs



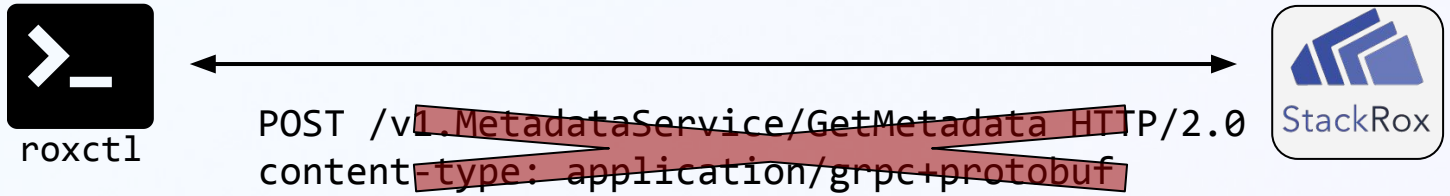
Functional Requirements

- Replace gRPC in roxctl CLI with *something* that works via HTTP/1.1
  - Bonus points for finding a *something* that is equally awesome

- Compatibility requirements:
  - **New** versions of **both** CLI and server should work through gRPC-incompatible LBs
  - **New/old** and **old/new** combinations should work through (at least) gRPC-*compatible* LBs
    - Can't use different RPC protocol altogether!

First Idea: Use REST APIs

- Most gRPC APIs have REST equivalents thanks to `grpc-gateway`



- Idea: use `protoc-gen-swagger` + `swagger-codegen`-generated bindings
  - Meh: lots of code changes, different data structures everywhere (no protobufs), buggy generated code

Some Background: Why gRPC Uses HTTP/2

- Some differences between HTTP/2 and HTTP/1.1:
  - **Binary** format (performance)
  - **Multiplexing** various streams onto one TCP connection (performance)
  - Full-duplex **client streaming** (functionality)
  - Guaranteed support for **trailers** (functionality, somewhat)

- … plus other stuff, such as server push (not relevant here)

HTTP/2 In-Depth: Client Streaming

- HTTP/1:
  - Client sends request (headers + body, possibly empty)
  - **Then,** server sends response (headers + body, possibly empty)

- HTTP/2:
  - Client sends request headers
  - **All** of the following may take place **simultaneously**:
    - Client sends request body (if any) in chunks/streaming
    - Server sends response headers, followed by response body (if any) in chunks/streaming
  - **Then,** server sends **trailers** (if any) and closes stream.

HTTP/2 In-Depth: Client Streaming

- Interleaved request flow between server and client
  - HTTP/1: Client talks (and finishes), then server responds
  - HTTP/2: Client *starts* talking, server may start responding any time (before client finishes)
- Required for client-streaming and bidi-streaming RPC calls
- Client streaming **cannot** be emulated in HTTP/1

- Good news: `roxctl` CLI only uses unary RPC calls 😅

HTTP/2 In-Depth: Trailers

- Trailers: metadata sent by server after response body
  - Think deferred headers
  - Typical use case: checksum of response body
  - gRPC use case: status/error information (`Grpc-Status`, `Grpc-Message`)

- Good news: trailers *can* be emulated in HTTP/1.1 😅
  - Just include trailer data as part of response **body**

gRPC-Web

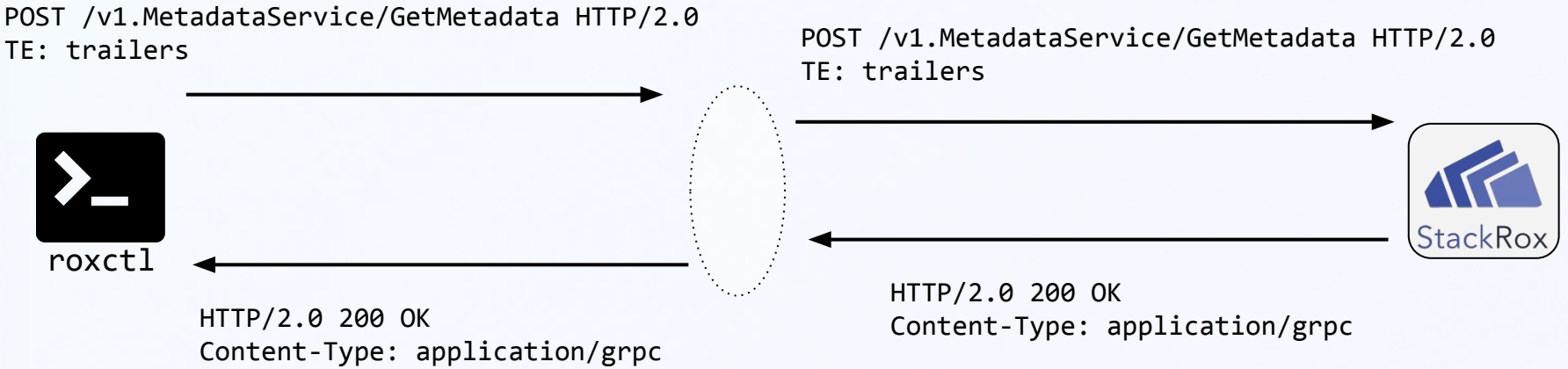
- Alternative (**non-HTTP2**) transport spec for gRPC geared towards web clients (browsers)
- Supports **unary and server-streaming** RPCs (no client or bidi-streaming)
- Encodes **trailers** as part of the response body (specially encoded message)

So gRPC-Web Solves All Our Problems, Right?

- Unfortunately, not quite:
  - No Golang-based client library
  - Can’t rely on Envoy proxy to be present in customer setup
  - Use of other proxies breaks compatibility requirements

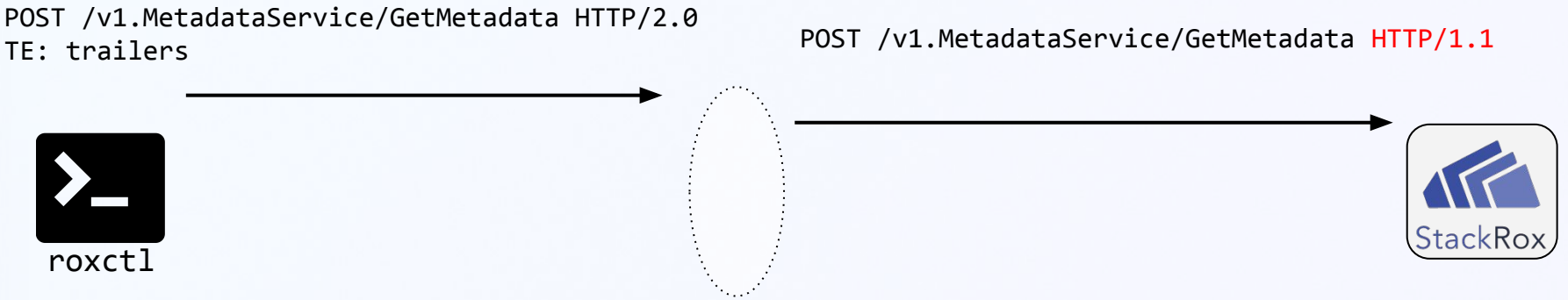
Solution: Automatic gRPC-Web Downgrading

- **Client** indicates ability to receive gRPC-Web response via `Accept` header
- **Server** infers need to send gRPC-Web response by HTTP protocol major version and TE (transfer encoding) header



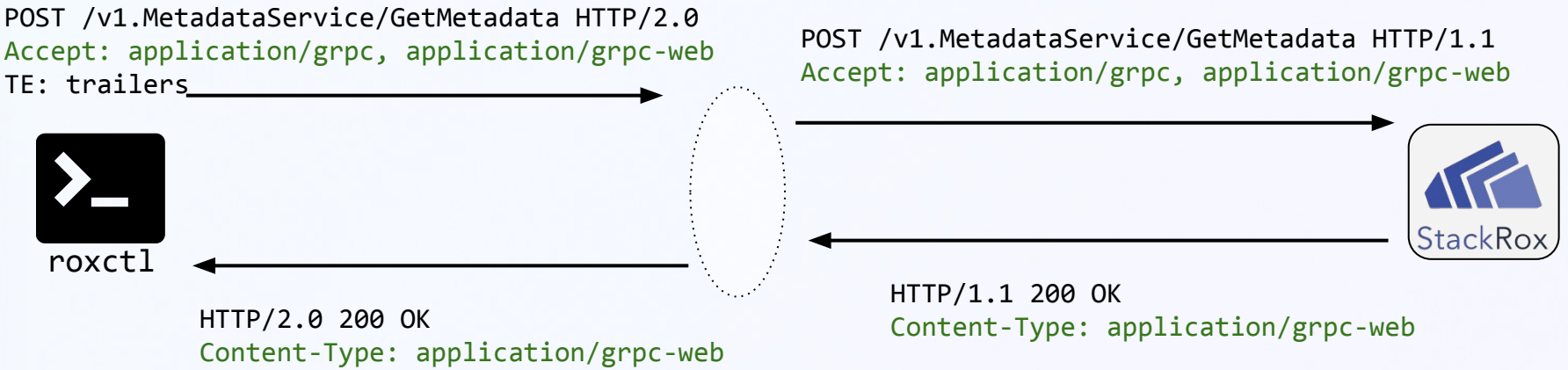
Solution: Automatic gRPC-Web Downgrading

- **Client** indicates ability to receive gRPC-Web response via `Accept` header
- **Server** infers need to send gRPC-Web response by HTTP protocol major version and TE (transfer encoding) header



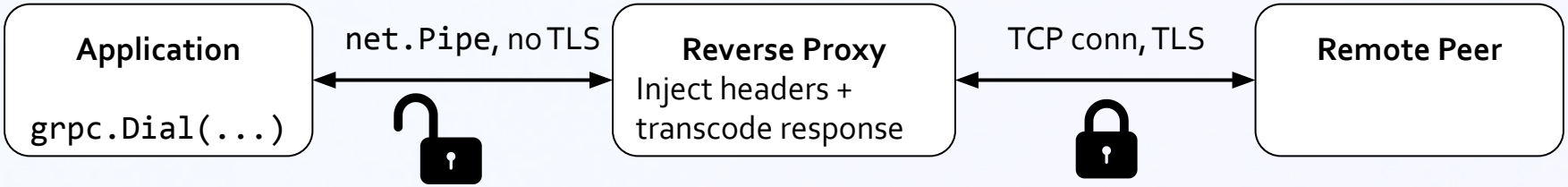
Solution: Automatic gRPC-Web Downgrading

- **Client** indicates ability to receive gRPC-Web response via `Accept` header
- **Server** infers need to send gRPC-Web response by HTTP protocol major version and TE (transfer encoding) header



Architecture / Implementation - Client

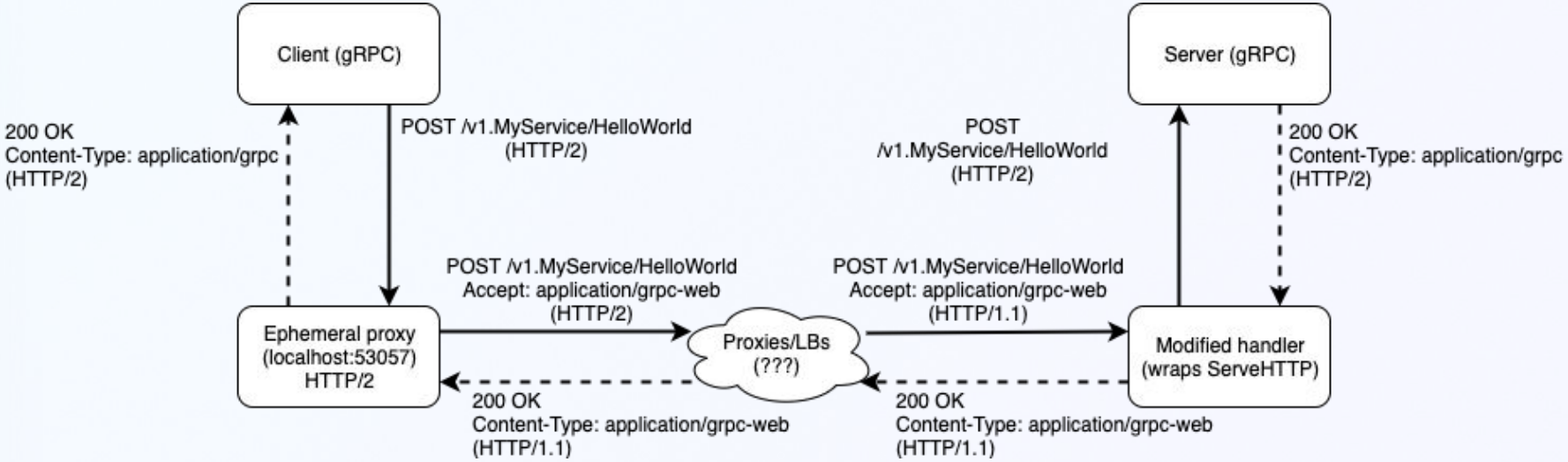
- Client-side: connections go to *local* reverse proxy
  - Outgoing requests: inject `Accept` header
  - Incoming responses: transcode gRPC-Web to gRPC-over-HTTP/2 for `application/grpc-web` content types
    - Effectively: trailer message → HTTP/2 trailers
  - TLS only between reverse proxy and remote peer



Architecture / Implementation - Server

- Server-side: wrap `(*grpc.Server).ServeHTTP`
  - Standardized Go HTTP(2) server handler
  - **If** `Accept: application/grpc-web` header is *present* **and** `TE: trailers` header is *absent*:
    - Modify request to spoof HTTP/2 and `TE: trailers` property
    - Intercept response writer to transcode HTTP/2 trailers into trailers message (part of response body)
    - Delegate to `(*grpc.Server).ServeHTTP`

Architecture / Implementation - Comprehensive View



Open-Source Library

- I want to use this in my application - can I?
  - **Yes**, if you use Go!
  - `go get golang.stackrox.io/grpc-http1`
  - Source code at https://github.com/stackrox/go-grpc-http1 (Apache 2 license)

Library Usage - Client Side

```
var tlsConf *tls.Config = ...
connectOpts := []grpc.DialOption{...}

conn, err := grpc.DialContext(
  context.Background(),
  "example.com:8443",
  grpc.WithTransportCredentials(credentials.NewTLS(tlsConf)),
  connectOpts...)

echoSvcClient := echo.NewEchoClient(conn)
echoSvcClient.UnaryEcho(context.Background(), &echo.EchoRequest{...})

...

defer conn.Close()
```

Library Usage - Client Side

```
import "golang.stackrox.io/grpc-http1/client"


var tlsConf *tls.Config = ...
connectOpts := []grpc.DialOption{...}

conn, err := client.ConnectViaProxy(
  context.Background(),
  "example.com:8443",
  tlsConf, // or nil + grpc.WithInsecure() for plaintext,
  connectOpts...)

echoSvcClient := echo.NewEchoClient(conn)
echoSvcClient.UnaryEcho(context.Background(), &echo.EchoRequest{...})

defer conn.Close()  // closes connection and shuts down proxy
```

Library Usage - Server Side

```
var tlsConf *tls.Config = ...

grpcSrv := grpc.NewServer()
echo.RegisterEchoServer(grpcSrv, myEchoSrv)

grpcSrv.Serve(tls.Listen("tcp", ":8443", tlsConf))
```

… and that's it!

Library Usage - Server Side

```
import "golang.stackrox.io/grpc-http1/server"

var tlsConf *tls.Config = ...

grpcSrv := grpc.NewServer()
echo.RegisterEchoServer(grpcSrv, myEchoSrv)

httpSrv := &http.Server{
  Handler: server.CreateDowngradingHandler(grpcSrv, nil),
  // Instead of nil, can also use another http.Handler
  // for multiplexed gRPC(-Web)/HTTP usage.
}
httpSrv.Serve(tls.Listen("tcp", ":8443", tlsConf))
```

... and that's it!

Outlook & Future Work

- `roxctl` is not performance-sensitive (at a per-RPC level), so performance optimizations are TBD
  - Idea: auto-sense connection capabilities to adaptively bypass proxy

- Client-streaming and bidi-streaming RPCs unsupported
  - Idea: use WebSockets for these
  - Coming soon!

Thanks for attending!

mi@stackrox.com - https://github.com/stackrox/go-grpc-http1