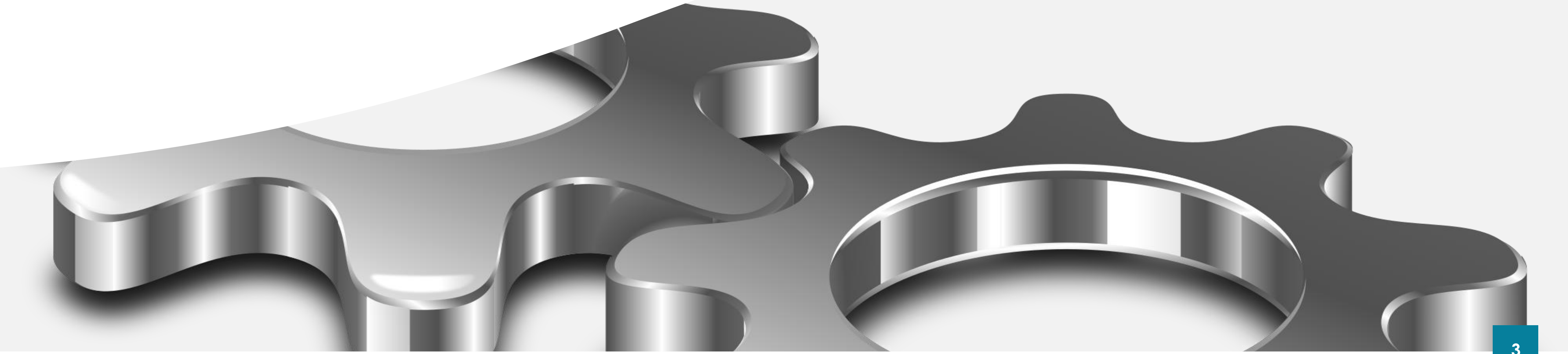
Golang Programming

Building Web Services with gRPC

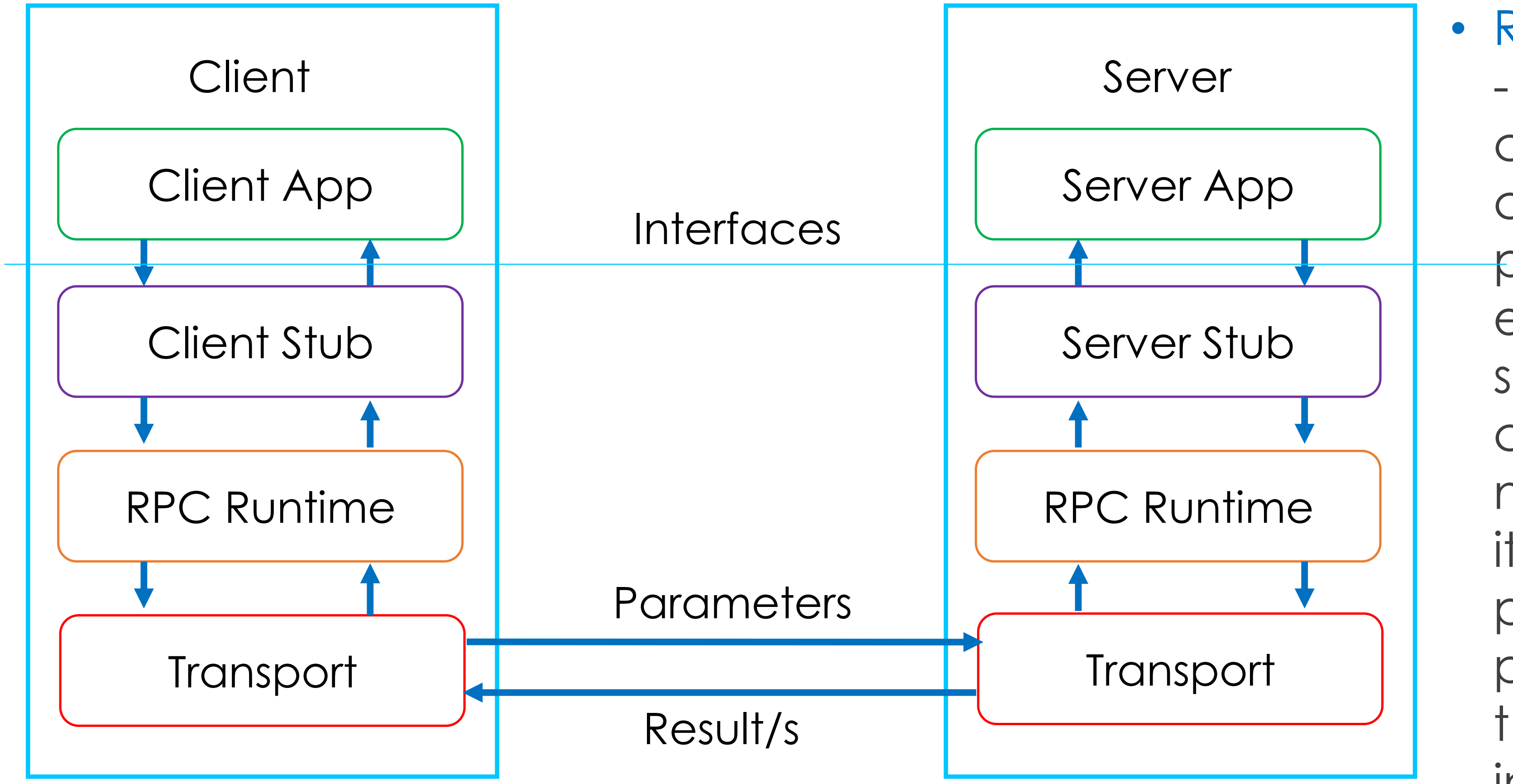
Where to Find The Code and Materials?

https://github.com/iproduct/coursego

Remote Procedure Calls (RPC) and gRPC



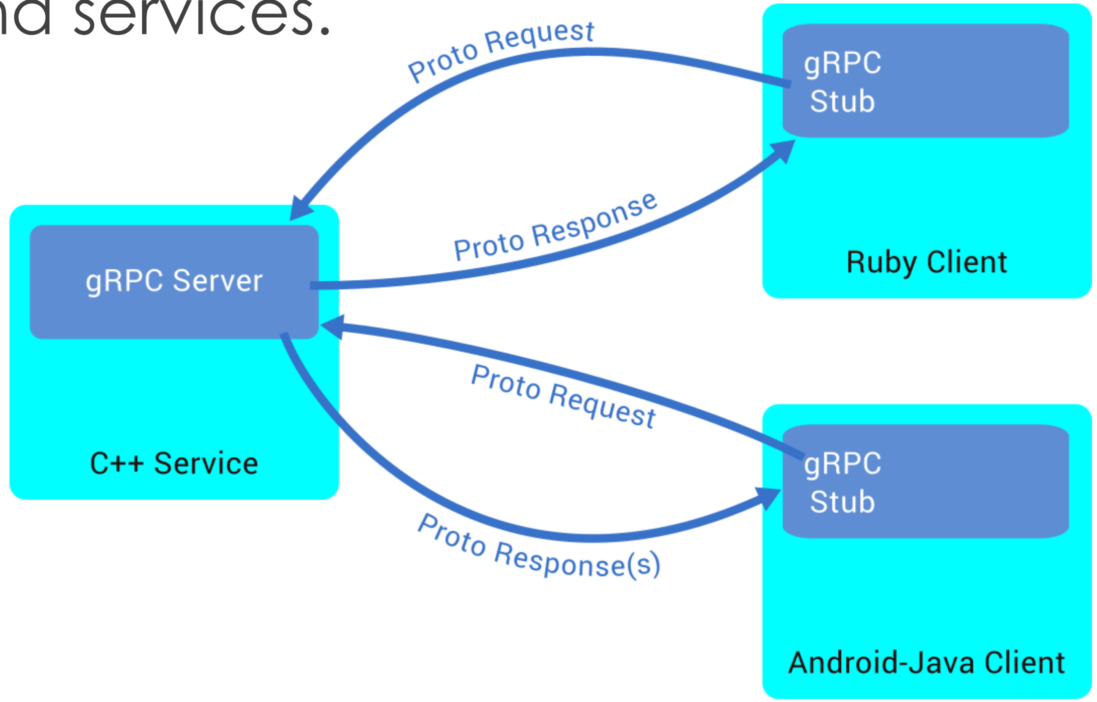
Remote Procedure Call (RPC)



- Remote Procedure Call (RPC) - a form of inter-process communication (IPC), when a computer program causes a procedure (subroutine) to execute in a different address space (commonly on another computer on a shared network), which is coded as if it were a normal (local) procedure call, without the programmer explicitly coding the details for the remote interaction.

gRPC (gRPC Remote Procedure Calls)

- gRPC is a modern open source high performance RPC framework that can run in any environment. It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication. It is also applicable in last mile of distributed computing to connect devices, mobile applications and browsers to backend services.



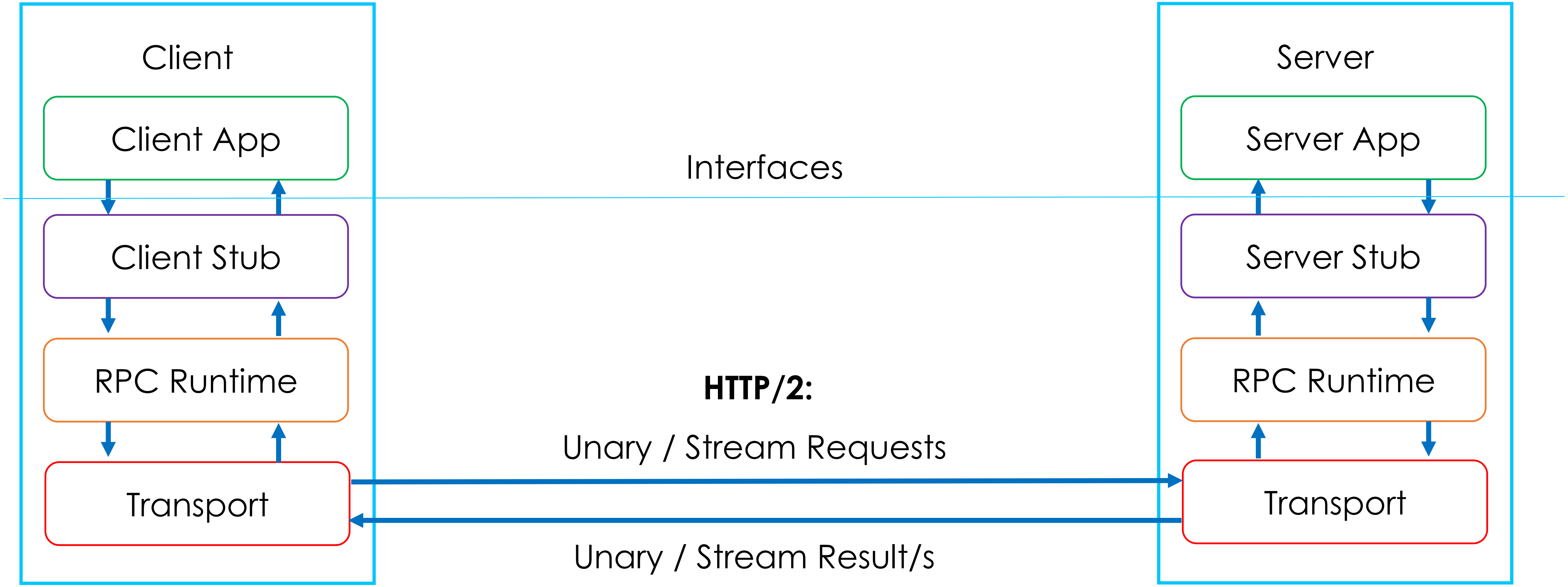
Main Usage Scenarios

- **Microservices** – designed for low latency and high throughput communication, lightweight and efficient polyglot microservices
- **Real-time P2P communication**: bi-directional streaming, client and server push in real-time, "last mile" (mobile, web, and Internet of Things).
- **Connecting mobile devices, browser clients** to backend services, generating efficient client libraries
- **Efficient network transport (constrained environments)**: messages are serialized with lightweight message format called Protobuf – messages are smaller than an JSON equivalents.
- **Inter-process communication (IPC)**: IPC transports such as Unix domain sockets / named pipes can be used with gRPC to communicate between apps on same machine - see Inter-process communication with gRPC.

gRPC Core Features

- Idiomatic client libraries in more than 12 languages – C/C++, C#, Dart, Go, Java, Kotlin, Node.js, Objective-C, PHP, Python, Ruby
- Highly efficient on wire and with a simple service definition framework using Protocol Buffers standard
- Bi-directional streaming with http/2 based transport
- Pluggable auth, tracing, load balancing and health checking using unary and stream client and server interceptors

gRPC (gRPC Remote Procedure Calls)



Who Uses gRPC ? [https://grpc.io/about/]

gRPC Design Principles - I

- Efficiency, security, reliability and behavioral analysis: Stubby -> SPDY, HTTP/2, QUIC (HTTP/3 – UDP based) public standards
- Services not Objects, Messages not References - promote microservices design philosophy of coarse-grained message exchange, avoiding the pitfalls of distributed objects and the fallacies of ignoring the network.
- Coverage & Simplicity - stack available on every popular platgorm, viable for CPU and memory-limited devices.
- Free & Open – open-source with licensing that should facilitate adoption.
- Interoperability & Reach - wire protocol surviving internet traversal.
- General Purpose & Performant – supporting broad class of use-cases.

Source: gRPC official blog - https://grpc.io/blog/principles/

gRPC Design Principles - II

- Layered - key facets of the stack must be able to evolve independently. A revision to the wire-format should not disrupt application layer bindings.
- Payload Agnostic - suspports different message types and encodings such as protocol buffers, JSON, XML, and Thrift; pluggable compression.
- Streaming – Storage systems rely on streaming and flow-control to express large data-sets. Other services, like voice-to-text or stock-tickers, rely on streaming to represent temporally related message sequences.
- Blocking & Non-Blocking – support asynchronous and synchronous processing of the sequence of messages exchanged by a client and server.
- Cancellation & Timeout – long-lived - cancellation allows servers to reclaim resources, cancellation can cascade; client timeout for a call.

Source: gRPC official blog - https://grpc.io/blog/principles/

gRPC Design Principles - III

- Lamedducking – graceful server shutdown: rejecting new, in-flight completed.
- Flow Control – allows for better buffer management and DOS protection.
- Pluggable – security, health-checking, load-balancing, failover, monitoring, tracing, logging, and so on; extensions points for plugging-in these features.
- Extensions as APIs – favor APIs rather than protocol extensions (health-checking, service introspection, load monitoring, and load-balancing).
- Metadata Exchange – cross-cutting concerns like authentication or tracing rely on the exchange of data that is not part of the declared interface.
- Standardized Status Codes – clients typically respond to errors returned by API calls in a limited number of ways; metadata exchange mechanism.

Source: gRPC official blog - https://grpc.io/blog/principles/

Protocol Buffers [https://github.com/protocolbuffers/]

- Protocol Buffers (Protobuf) – method for serializing structured data, involves an interface description language that describes the structure of some data and a program that generates source code from that description for generating or parsing a stream of bytes that represents the structured data.
- Smaller and faster than XML and JSON
- Basis for a custom remote procedure call (RPC) system that is used for nearly all inter-machine communication at Google.
- Similar to the Apache Thrift (used by Facebook), Ion (created by Amazon), or Microsoft Bond protocols, offering as well a concrete RPC protocol stack to use for defined services – gRPC.
- Install compiler from: https://github.com/protocolbuffers/protobuf/releases/

Source: Wikipedia - https://en.wikipedia.org/wiki/Protocol_Buffers

Protocol Buffers [https://developers.google.com/protocol-buffers/docs/proto3]

- Data structures (messages) and services are described in a proto definition file (.proto) and compiled with **protoc**. This compilation generates code that can be invoked by a sender or recipient of these data structures. E.g. example.proto -> example.pb.go and example_grpc.pb.go. They define Golang types and methods for each message and service in example.proto.

```
protoc --go_out=. --go-grpc_out=. --go_opt=paths=source_relative  \
          --go-grpc_opt=paths=source_relative *.proto
```

```
go get google.golang.org/protobuf/cmd/protoc-gen-go \
google.golang.org/grpc/cmd/protoc-gen-go-grpc

mkdir -p generated

protoc --proto_path=api/proto/v1 --proto_path=third_party --go_out=./generated  \
--go-grpc_out=./generated todo_service.proto
```

More information in Wikipedia: https://en.wikipedia.org/wiki/Protocol_Buffers

Programming with gRPC in 3 Simple Steps:

1. Define a service in a .proto file.
2. Generate server and client code using the protocol buffer compiler – protoc.
3. Use the Go gRPC API to write a simple client and server for your service.

Source: Wikipedia - https://en.wikipedia.org/wiki/Protocol_Buffers

Example: helloworld.proto

```
syntax = "proto3";

option go_package = "google.golang.org/grpc/examples/helloworld/helloworld";

package helloworld;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}
// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}
// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

Example: helloworld server.go

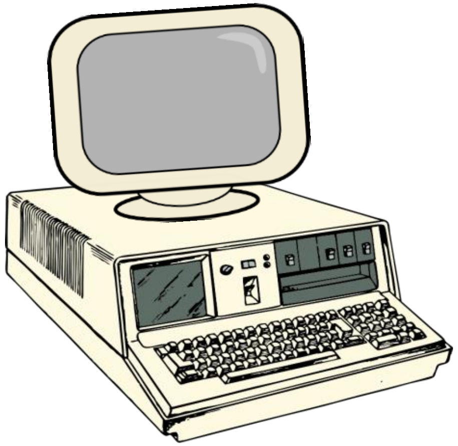
```go
package main
import ("context"; pb "github.com/iproduct/coursego/10-grpc-hello/helloworld"; "google.golang.org/grpc"; "log";"net")
const port = ":50051"

// server is used to implement helloworld.GreeterServer.
type server struct { pb.UnimplementedGreeterServer }

// SayHello implements helloworld.GreeterServer
func (s *server) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.HelloReply, error) {
		log.Printf("Received: %v", in.GetName())
		return &pb.HelloReply{Message: "Hello " + in.GetName()}, nil
}
func main() {
		lis, err := net.Listen("tcp", port)
		if err != nil { log.Fatalf("failed to listen: %v", err) }
		s := grpc.NewServer()
		pb.RegisterGreeterServer(s, &server{})
		if err := s.Serve(lis); err != nil { log.Fatalf("failed to serve: %v", err)		}
}
```

Example: helloworld client.go

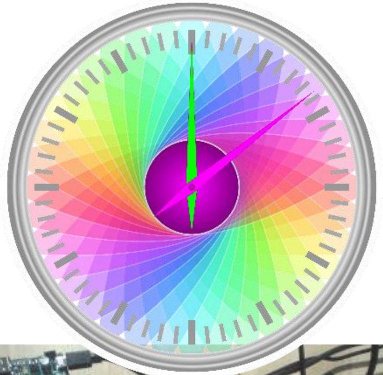
```
package main
import ( "context"; pb "github.com/iproduct/coursego/10-grpc-hello/helloworld"; "log"; "os"; "time";
  "google.golang.org/grpc" )
const ( address     = "localhost:50051“; defaultName = "world“ )

func main() {
		// Set up a connection to the server.
		conn, err := grpc.Dial(address, grpc.WithInsecure(), grpc.WithBlock())
		if err != nil { log.Fatalf("did not connect: %v", err) }
		defer conn.Close()
		c := pb.NewGreeterClient(conn)
		// Contact the server and print out its response.
		name := defaultName
		if len(os.Args) > 1 { name = os.Args[1] }
		ctx, cancel := context.WithTimeout(context.Background(), time.Second)
		defer cancel()
		r, err := c.SayHello(ctx, &pb.HelloRequest{Name: name})
		if err != nil { log.Fatalf("could not greet: %v", err) }
		log.Printf("Greeting: %s", r.GetMessage())
}
```

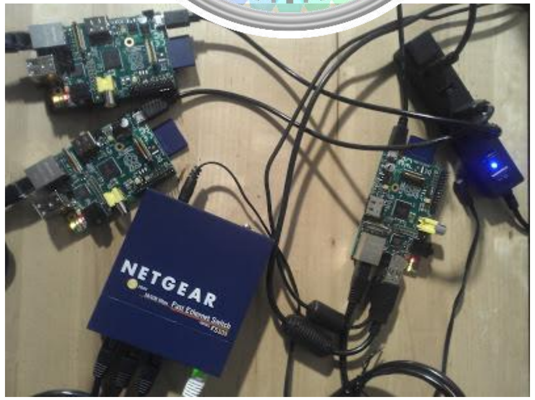
Data Streaming with gRPC

Need for Speed: Data Streaming

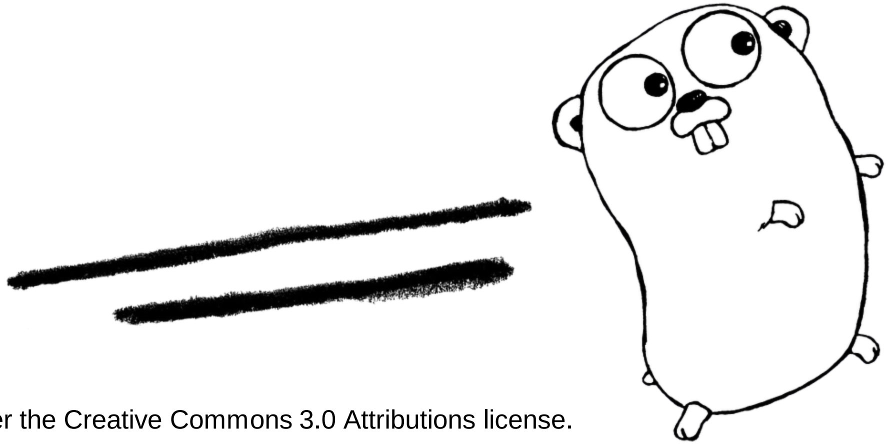


hadoop



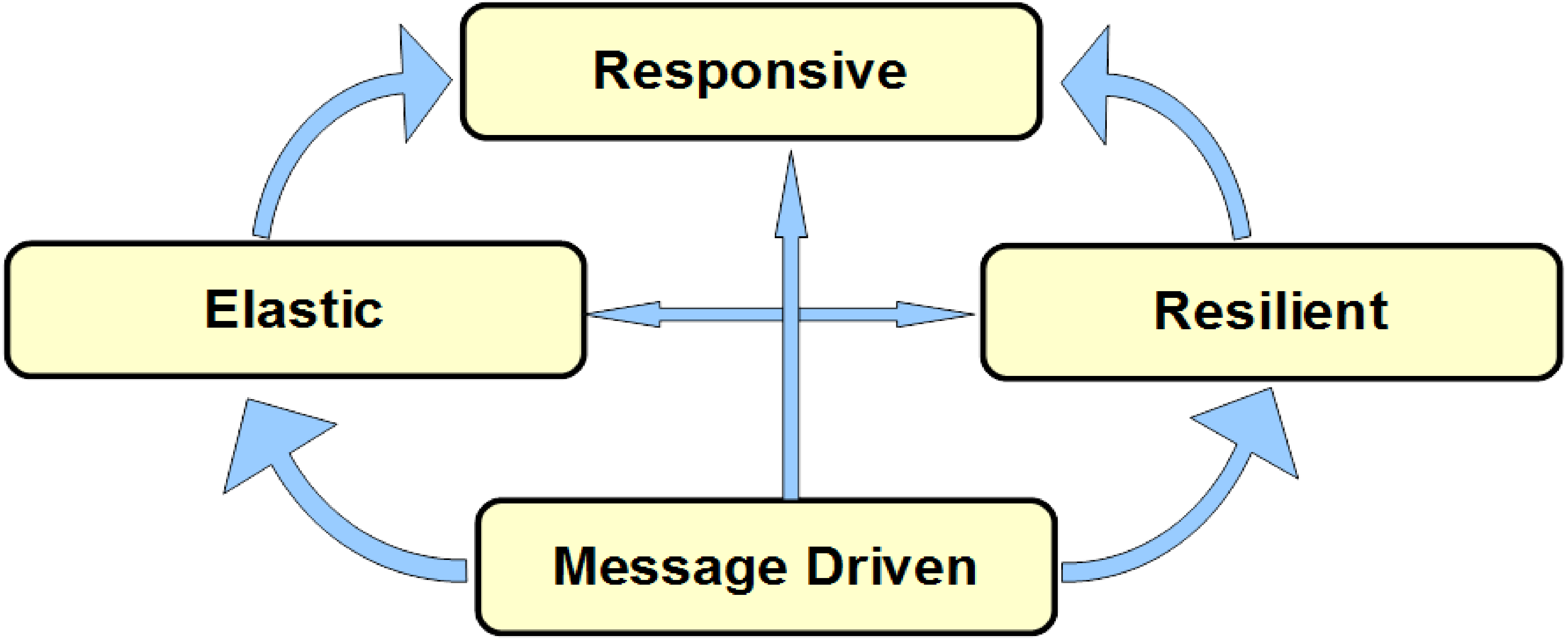


?



Reactive Manifesto

http://www.reactivemanifesto.org



Data / Event / Message Streams

“Conceptually, a stream is a (potentially never-ending) flow of data records, and a transformation is an operation that takes one or more streams as input, and produces one or more output streams as a result.”

Apache Flink: Dataflow Programming Model

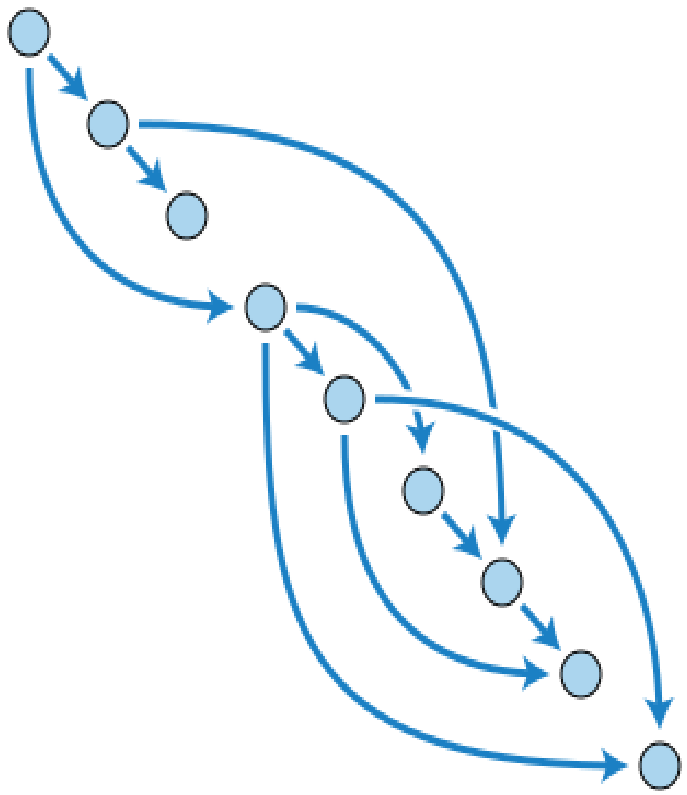
Data Stream Programming

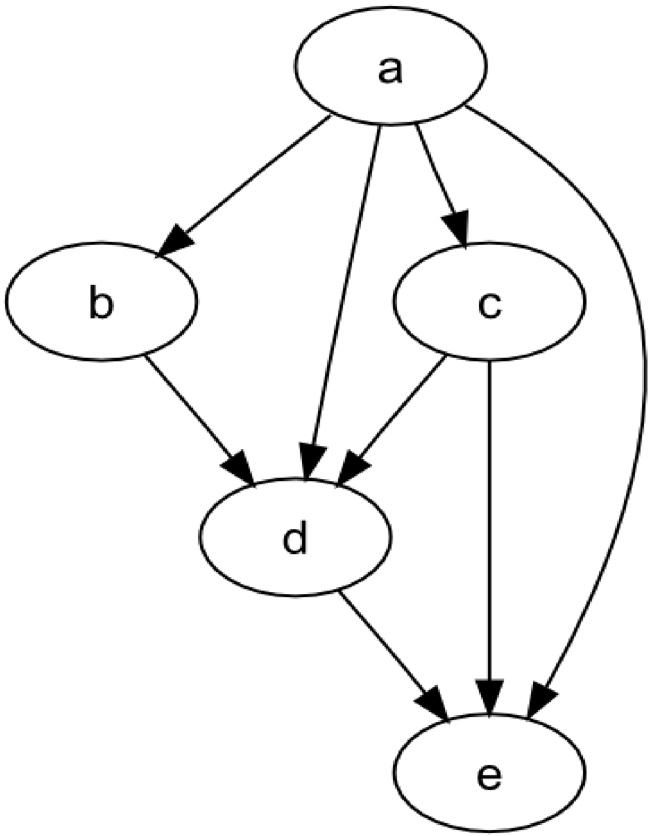
The idea of abstracting logic from execution is hardly new -- it was the dream of SOA. And the recent emergence of microservices and containers shows that the dream still lives on.

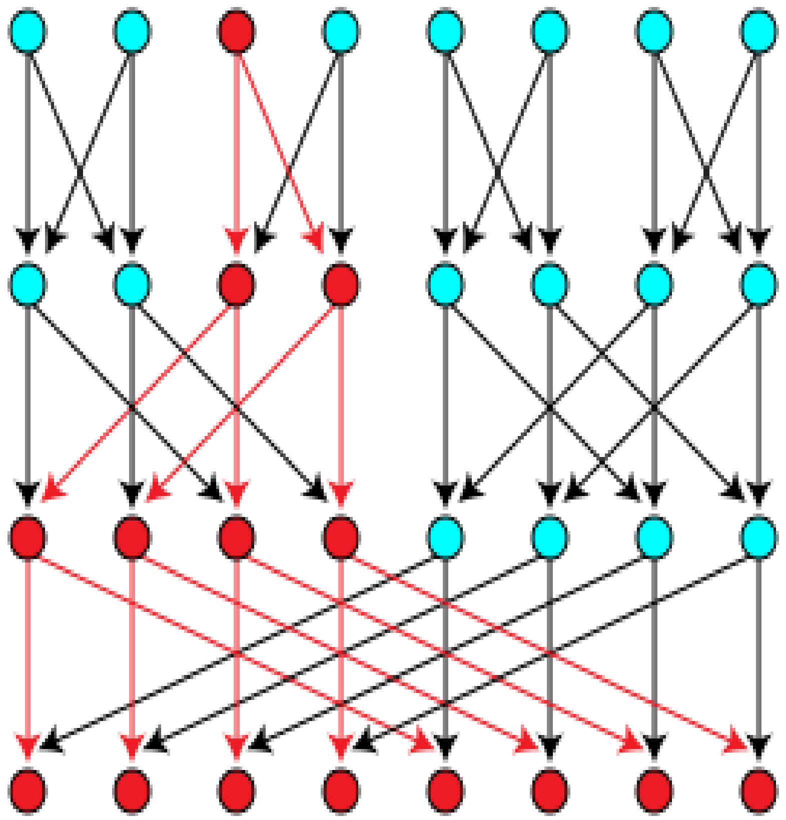
For developers, the question is whether they want to learn yet one more layer of abstraction to their coding. On one hand, there's the elusive promise of a common API to streaming engines that in theory should let you mix and match, or swap in and swap out.

Tony Baer (Ovum) @ ZDNet - Apache Beam and Spark: New coopetition for squashing the Lambda Architecture?

Direct Acyclic Graphs - DAG



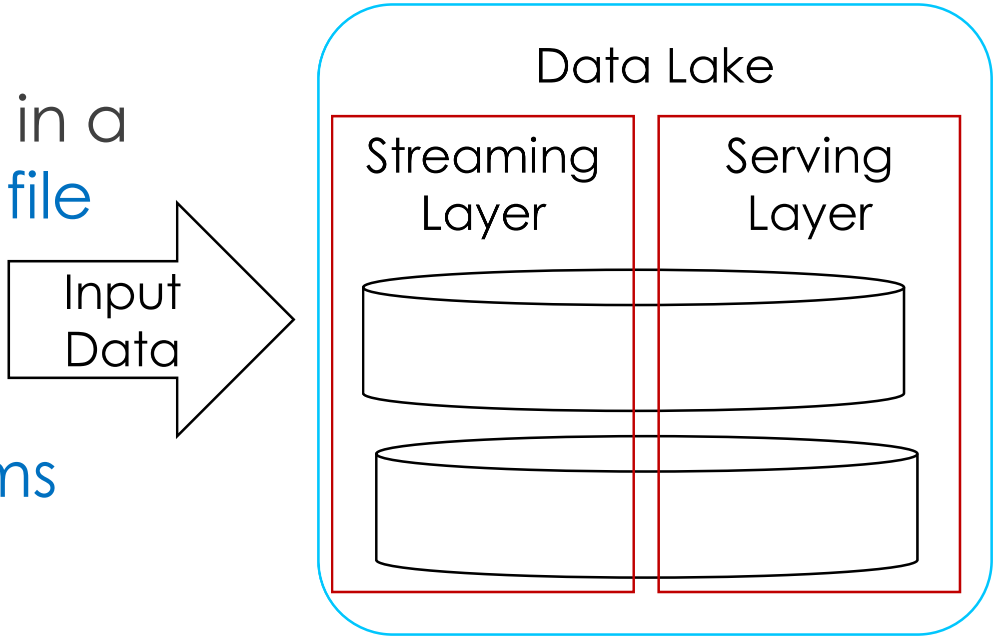




Kappa Architecture II

Query = K (New Data) = K (Live streaming data)

- Multiple data events or queries are logged in a queue to be catered against a distributed file system storage or history.
- The order of the events and queries is not predetermined. Stream processing platforms can interact with database at any time.
- It is resilient and highly available as handling terabytes of storage is required for each node of the system to support replication.
- Machine learning is done on the real time basis



Types of Streaming with gRPC – I

- **Simple RPC (no streaming)** – client sends a request to the server using the stub and waits for a response to come back, just like a normal function call:

// Obtains the feature at a given position.
rpc GetFeature(Point) **returns** (Feature) {}

- **Server-side streaming RPC** – client sends a request to the server and gets a stream to read a sequence of messages back. The client reads from the returned stream until there are no more messages - **stream** keyword before the *response* type:

// Obtains the Features available within the given Rectangle. Results are // streamed rather than returned at once (e.g. in a response message with a // repeated field), as the rectangle may cover a large area and contain a // huge number of features.
rpc ListFeatures(Rectangle) **returns** (**stream** Feature) {}

Source: gRPC official blog - https://grpc.io/blog/principles/

Types of Streaming with gRPC – II

- **Client-side streaming RPC** – client writes a sequence of messages and sends them to the server, again using a provided stream. Once the client has finished writing the messages, it waits for the server to read them all and return its response – **stream** keyword before the request type:

// Accepts a stream of Points on a route being traversed, returning a // RouteSummary when traversal is completed.
rpc RecordRoute(**stream** Point) **returns** (RouteSummary) {}

- **Bidirectional streaming RPC** – both sides send a sequence of messages using a read-write stream. The two streams operate independently, so clients and servers can read and write in whatever order they like: for example, the server could wait to receive all the client messages before writing its responses, or it could alternately read a message then write a message, or some other combination of reads and writes. The order of messages in each stream is preserved – **stream** keyword before both the request and the response:

// Accepts a stream of RouteNotes sent while a route is being traversed, // while receiving other RouteNotes (e.g. from other users).
rpc RouteChat(**stream** RouteNote) **returns** (stream RouteNote) {}

Source: gRPC official blog - https://grpc.io/blog/principles/

Example – Bidirectional Streaming: math.proto

```
syntax = "proto3";

package protobuf;

option go_package = "./protomath";

service Math {
  rpc Max (stream Request) returns (stream Response) {}
}

message Request {
  int32 num = 1;
}

message Response {
  int32 result = 1;
}
```

Example – Bidirectional Streaming: server.go - I

```
type server struct{
        pm.Math_MaxServer
        pm.UnimplementedMathServer
}
func (s server) Max(srv pm.Math_MaxServer) error {
        log.Println("start new server")
        var max int32
        ctx := srv.Context()
        for {
                select {    // exit if context is done or continue
                        case <-ctx.Done(): return ctx.Err()
                        default:
                }
                req, err := srv.Recv()    // receive data from stream
                if err == io.EOF {  // return will close stream from server side
                         log.Println("exit")
                         return nil
                }
                if err != nil { log.Printf("receive error %v", err); continue }
```

Example – Bidirectional Streaming: server.go - II

```
			// continue if number reveived from stream
			// less than max
			if req.Num <= max {
					continue
			}

			// update max and send it to stream
			max = req.Num
			resp := pm.Response{Result: max}
			if err := srv.Send(&resp); err != nil {
					log.Printf("send error %v", err)
			}
			log.Printf("send new max=%d", max)
		}
}
```

Example – Bidirectional Streaming: server.go - III

```go
func main() {
		// create listiner
		lis, err := net.Listen("tcp", ":50005")
		if err != nil {
			log.Fatalf("failed to listen: %v", err)
		}

		// create grpc server
		s := grpc.NewServer()
		pm.RegisterMathServer(s, server{})

		// and start...
		if err := s.Serve(lis); err != nil {
			log.Fatalf("failed to serve: %v", err)
		}
}
```

Example – Bidirectional Streaming : client.go - I

```
func main() {
        rand.Seed(time.Now().Unix())
        // dail server
        conn, err := grpc.Dial(":50005", grpc.WithInsecure())
        if err != nil {
                log.Fatalf("can not connect with server %v", err)
        }
         defer conn.Close()
        // create stream
        client := pm.NewMathClient(conn)
        stream, err := client.Max(context.Background())
        if err != nil {
                log.Fatalf("openn stream error %v", err)
        }
        var max int32
        ctx := stream.Context()
        done := make(chan bool)
```

Example – Bidirectional Streaming : client.go - II

```
// first goroutine sends random increasing numbers to stream and closes int after 20 iterations
go func() {
	for i := 1; i <= 20; i++ {
		// generate random nummber and send it to stream
		rnd := int32(rand.Intn(i))
		req := pm.Request{Num: rnd}
		if err := stream.Send(&req); err != nil {
			log.Fatalf("can not send %v", err)
		}
		log.Printf("%d sent", req.Num)
		time.Sleep(time.Millisecond * 200)
	}
	if err := stream.CloseSend(); err != nil {
		log.Println(err)
	}
}()
```

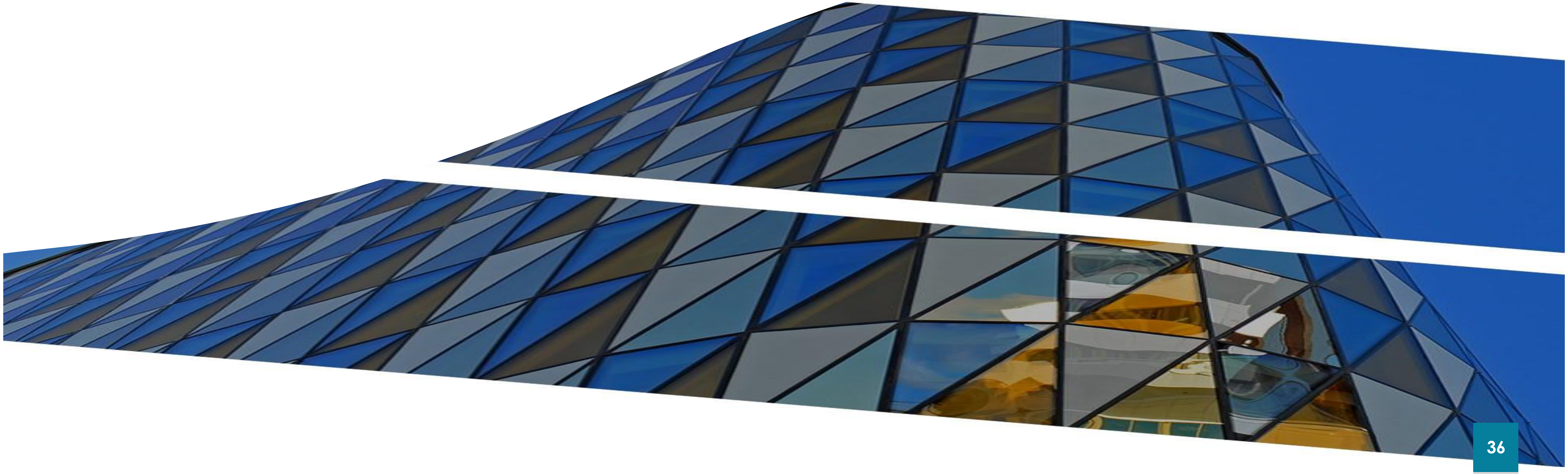
Example – Bidirectional Streaming : client.go - III

```go
// second goroutine receives data from stream and saves result in max variable
// if stream is finished it closes done channel
go func() {
	for {
		resp, err := stream.Recv()
		if err == io.EOF {
			close(done)
			return
		}
		if err != nil {
			log.Fatalf("can not receive %v", err)
		}
		max = resp.Result
		log.Printf("new max %d received", max)
	}
}()
```

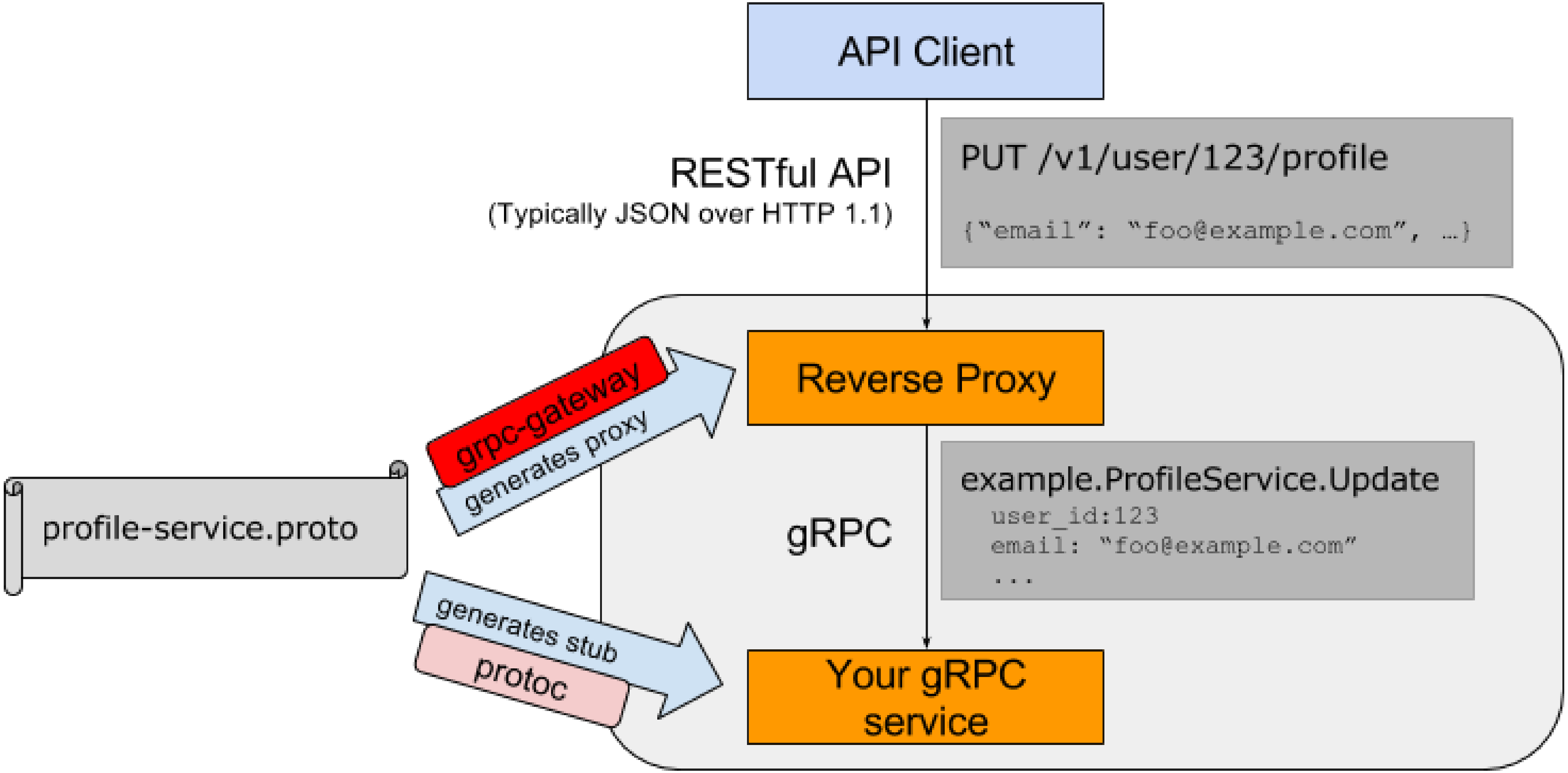
Example – Bidirectional Streaming : client.go - IV

```go
        // third goroutine closes done channel
        // if context is done
        go func() {
                <-ctx.Done()
                if err := ctx.Err(); err != nil {
                        log.Println(err)
                }
                close(done)
        }()

        <-done
        log.Printf("finished with max=%d", max)
}
```

And More …

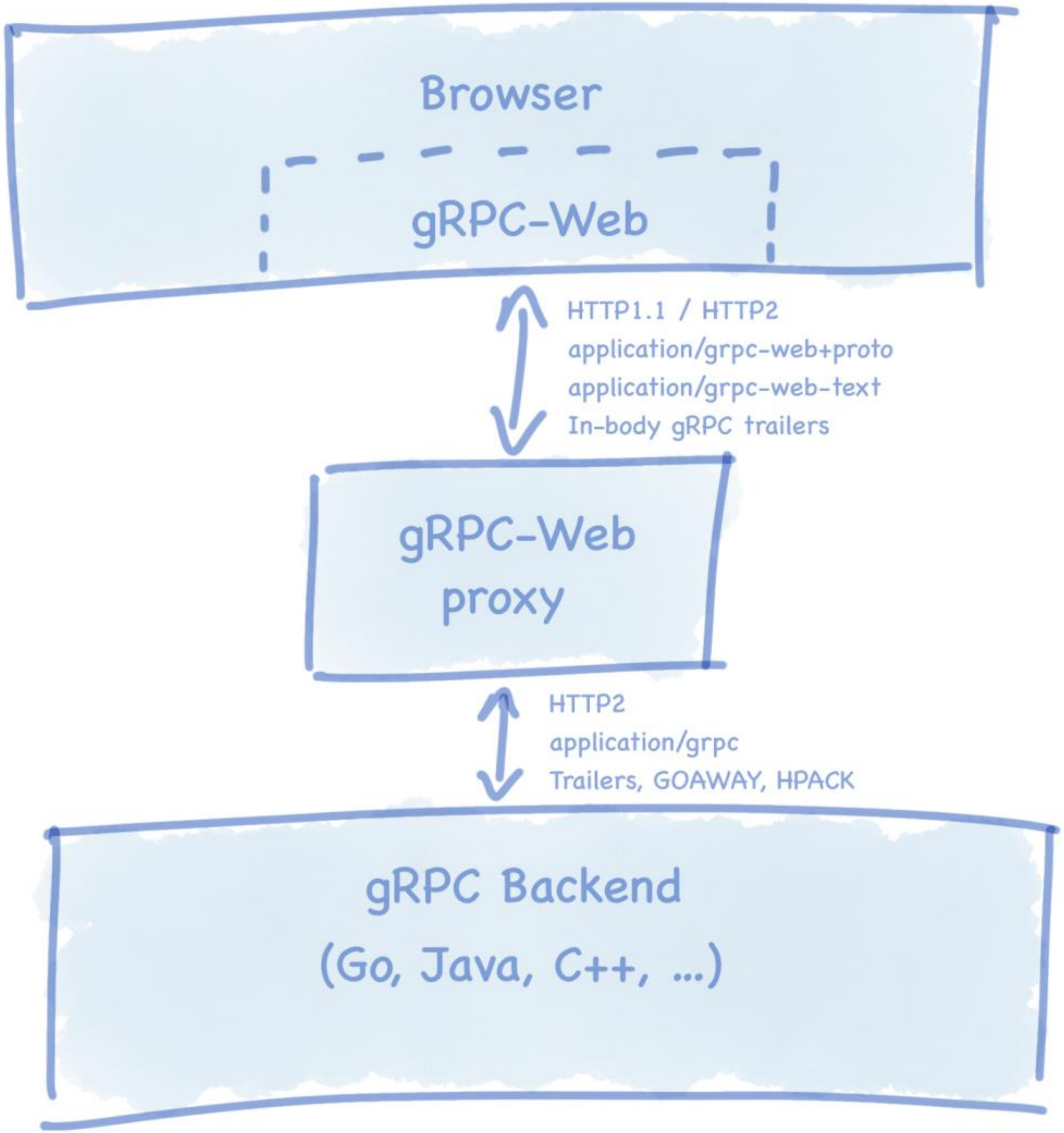


Exposing gRPC Service as REST Service



Source: Wikipedia – https://github.com/phuongdo/go-grpc-tutorial

gRPC: Web Clients



Source: *https://grpc.io/blog/state-of-grpc-web/*

Thank’s for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

http://robolearn.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD