Distributed Programming with Go

Concepts of Programming Languages
22 November 2018

Johannes Weigend (QAware GmbH)
University of Applied Sciences Rosenheim

Distributed Programming with Go

- Go supports all major network protocols within the standard library
- Channels and Go Routines simplify concurrent servers
- Typical samples can be found here:

github.com/tumregels/Network-Programming-with-Go (https://github.com/tumregels/Network-Programming-with-Go) 2

Some samples

- TCP/IP Sockets

github.com/tumregels/Network-Programming-with-Go/blob/master/socket/tcp_sockets.md (https://github.com/tumregels/Network-Programming-with-Go/blob/master/socket/tcp_sockets.md) 3

Distributed Computing is Hard!

Fallacies of distributed computing

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

Remote Procedure Calls

- Enable application to call functions on another process or machine
- Binary transfer of marshalled objects (-> Fast)
- Long history (Sun RPC (NFS), OMG Corba, DCE RPC, MS RPC, XML RPC, GRPC)
- Every major language has a proprietary RPC Framework (Go - net.rpc)
- Can be implemented on top of other protocols (HTTP/REST, HTTP/SOAP)

Introduction into GRPC

- Google RPC / GRPC is a language independent RPC based on Protocol Buffers
- Native implementations are available for Go, Java and C++
- Language bindings (by using C/C++) are available for many other Languages
- Major design decisions: Interoperability, Speed, HTTP based

www.youtube.com/watch?v=J-NTfvYL_OE (https://www.youtube.com/watch?v=J-NTfvYL_OE)

UUID Service with GRPC

In the following slides, we develop an UUID service which returns a network wide, unique id to a client.
We will separate the business logic from the GRPC code in a way to switch between a non distributed system and a distributed system with two processes (client/server)

UUID Service - Business API

```go
// Package idserv contains the IDService API.
package idserv

// IDService can be used to produce network wide unique ids.
type IDService interface {

    // NewUUID generates an UUID with a given client prefix.
    NewUUID(clientID string) string
}
```

- The Business API is a non technical API without any knowledge of distribution
- The API can be local tested (location transparent/independent)
- Local testability is strongly needed for non trivial functionality
- To enable distribution, methods should not use/return complex networks of objects

UUID Service - Client

```go
// GenerateIds calls n-times NewUUID() in a loop and returns the result as slice.
func GenerateIds(count int, service idserv.IDService) []string {
    result := make([]string, count)
    for i := 0; i < count; i++ {
        result[i] = service.NewUUID("c1")
    }
    return result
}
```

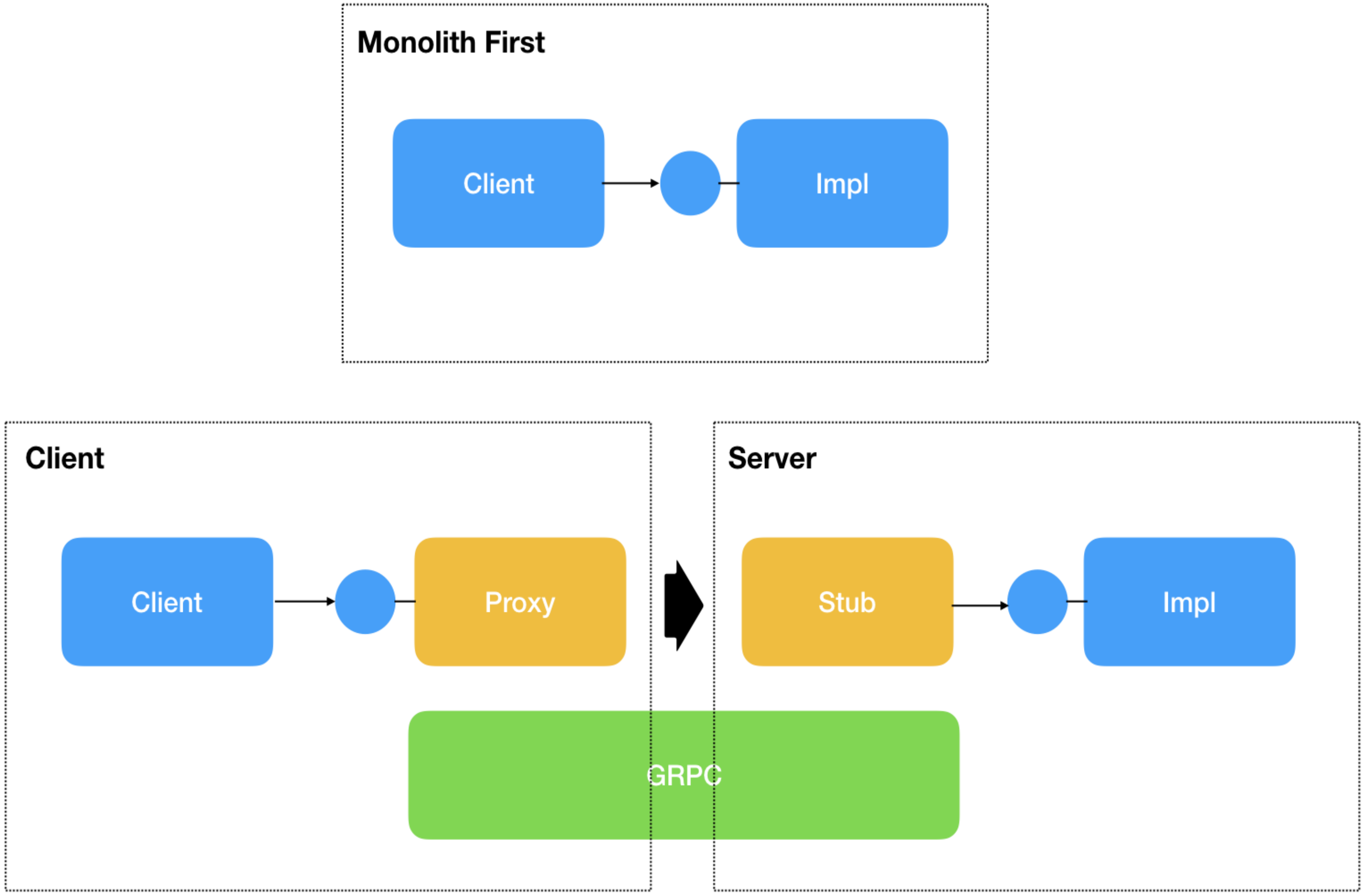
- The client logic has no idea which implementation is behind the API interface

UUID Service - Local Implementation (The Business Logic)

```go
type IDServiceImpl struct {
}

// The last given Id.
var lastID int64

// NewIDServiceImpl creates a new instance
func NewIDServiceImpl() *IDServiceImpl {
    return new(IDServiceImpl)
}

// NewUUID implements the IDService interface.
func (ids *IDServiceImpl) NewUUID(clientID string) string {
    result := atomic.AddInt64(&lastID, 1)
    return fmt.Sprintf("%v:%v", clientID, result)
}
```

- The implementation uses the atomic package to make it thread safe

Monolith First - The Proxy Pattern



Building a GRPC Proxy/Stub Combination

- The GRPC interface definition (Protocol Buffers (idserv.pb))

```
// The IDService definition
service IDService {
  // NewUUID generates a globally unique ID
  rpc NewUUID (IdRequest) returns (IdReply) {}
}

// The client sends a unique id.
message IdRequest {
  string clientId = 1;
}
```

- You have to generate the idserv.pb.go file with the GRPC protoc compiler (not part of the Go installation)
- The .pb.go file contains types and structures to implement and call the service

The Proxy implements the API interface on the client side (1/2)

```go
// Proxy is a client side proxy which encapsulates the RPC logic. It implements the IDService interface.
type Proxy struct {
    connection *grpc.ClientConn
}

// NewProxy creates a Proxy and starts the server connection
func NewProxy() *Proxy {
    p := new(Proxy)
    conn, err := grpc.Dial(address, grpc.WithInsecure())
    if err != nil {
        panic(fmt.Sprintf("did not connect: %v", err))
    }
    p.connection = conn
    return p
}

// NewUUID implements the IDService interface.
```

The Proxy implements the API interface on the client side (2/2)

```go
// NewUUID implements the IDService interface.
func (p *Proxy) NewUUID(clientID string) string {
    c := idserv.NewIDServiceClient(p.connection)
    ctx, cancel := context.WithTimeout(context.Background(), time.Second)
    defer cancel()
    r, err := c.NewUUID(ctx, &idserv.IdRequest{ClientId: clientID})
    if err != nil {
        log.Printf("could not generate id: %v", err)
        r.Uuid = ""
    }
    return r.Uuid
}
```

- The generated GRPC code differs from the local function call
- The proxy acts as adapter between the API and the GRPC interface
- Parameters, Return Values and Errors are different
- The Proxy maps all these to remote structures, errors or exceptions

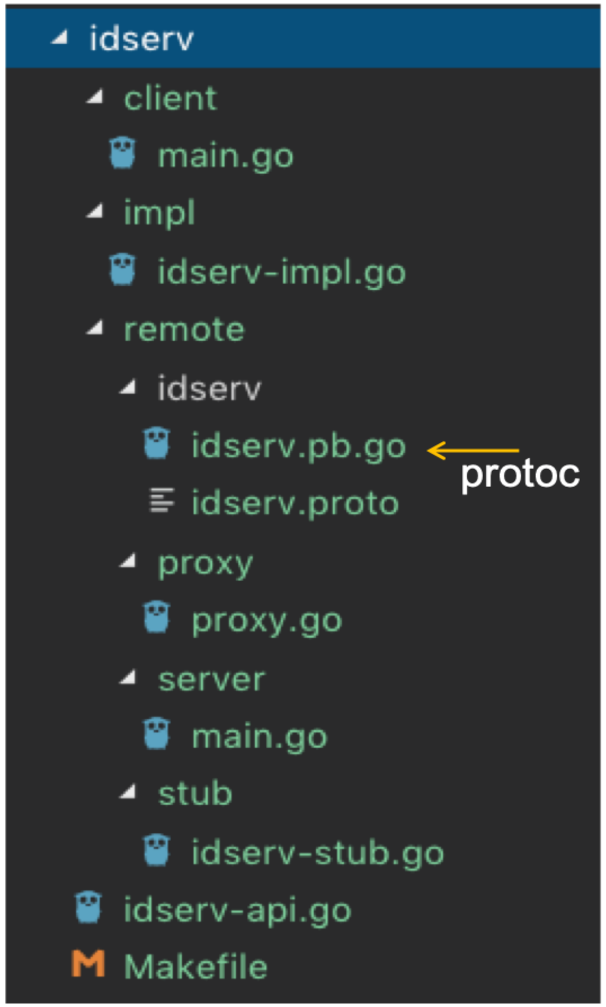
The Stub implements the GRPC interface and calls the Business API

```go
// Copyright 2018 Johannes Weigend
// Licensed under the Apache License, Version 2.0

package stub

import (
    "context"

    "github.com/jweigend/concepts-of-programming-languages/dp/idserv/core"
    "github.com/jweigend/concepts-of-programming-languages/dp/idserv/remote/idserv"
)

// Server is used to implement idserv.IdServer
type Server struct{}

// NewUUID implements idserv.IdService interface
func (s *Server) NewUUID(c context.Context, r *idserv.IdRequest) (*idserv.IdReply, error) {
    service := core.IDServiceImpl{}
    return &idserv.IdReply{Uuid: service.NewUUID(r.GetClientId())}, nil
}
```

The Server starts the Listener and registers the Stub

```go
func main() {
    lis, err := net.Listen("tcp", port)
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    s := grpc.NewServer()
    idserv.RegisterIDServiceServer(s, &stub.Server{})
    // Register reflection service on gRPC server.
    reflection.Register(s)
    if err := s.Serve(lis); err != nil {
        log.Fatalf("failed to serve: %v", err)
    }
}
```

The Directory Layout separates Logic from RPC code



```
idserv
  client
    main.go
  impl
    idserv-impl.go
  remote
    idserv
      idserv.pb.go
      idserv.proto
    proxy
      proxy.go
    server
      main.go
    stub
      idserv-stub.go
  idserv-api.go
  Makefile
```

Further Information

grpc.io/ (https://grpc.io/)

grpc.io/docs/quickstart/go.html (https://grpc.io/docs/quickstart/go.html)

github.com/grpc/grpc-go/tree/master/examples (https://github.com/grpc/grpc-go/tree/master/examples)

Thank you

Johannes Weigend (QAware GmbH)
University of Applied Sciences Rosenheim
johannes.weigend@qaware.de (mailto:johannes.weigend@qaware.de)
http://www.qaware.de (http://www.qaware.de)
@johannesweigend (http://twitter.com/johannesweigend)