Remote Procedure Call
gRPC

Илья Кокорин

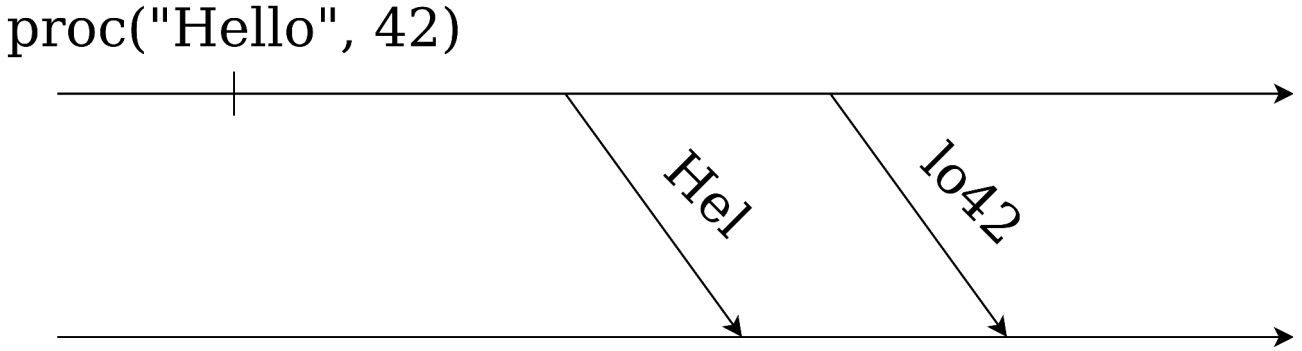
kokorin.ilya.1998@gmail.com

Постановка задачи

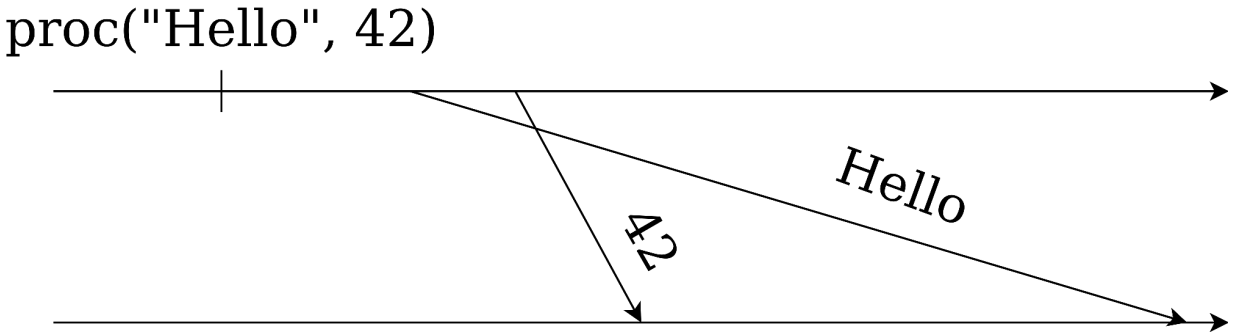
- Хотим вызвать функцию, реализация которой работает на другой машине
  - `result := RPC.Procedure(args...)`
- Хотим спрятать от пользователя:
  - Сериализацию параметров
  - Сериализацию возвращаемого значения
  - Переиспользование канала
  - Разбиение потока байт на сообщения (фрейминг)
  - Мультиплексирование
  - …

RPC: Проблемы при реализации

- Фрейминг
  - Для RPC поверх TCP

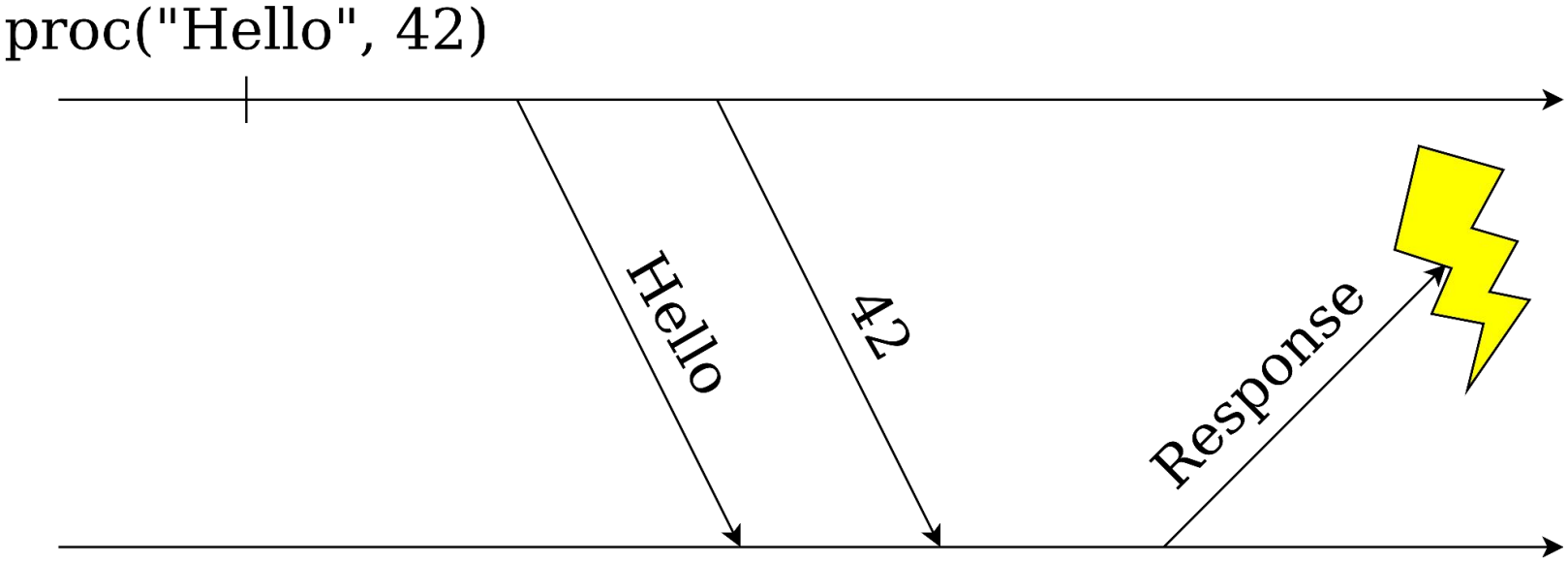


- Переупорядочивание сообщений
  - Для RPC поверх UDP



RPC: Проблемы при реализации

- Потеря сообщений

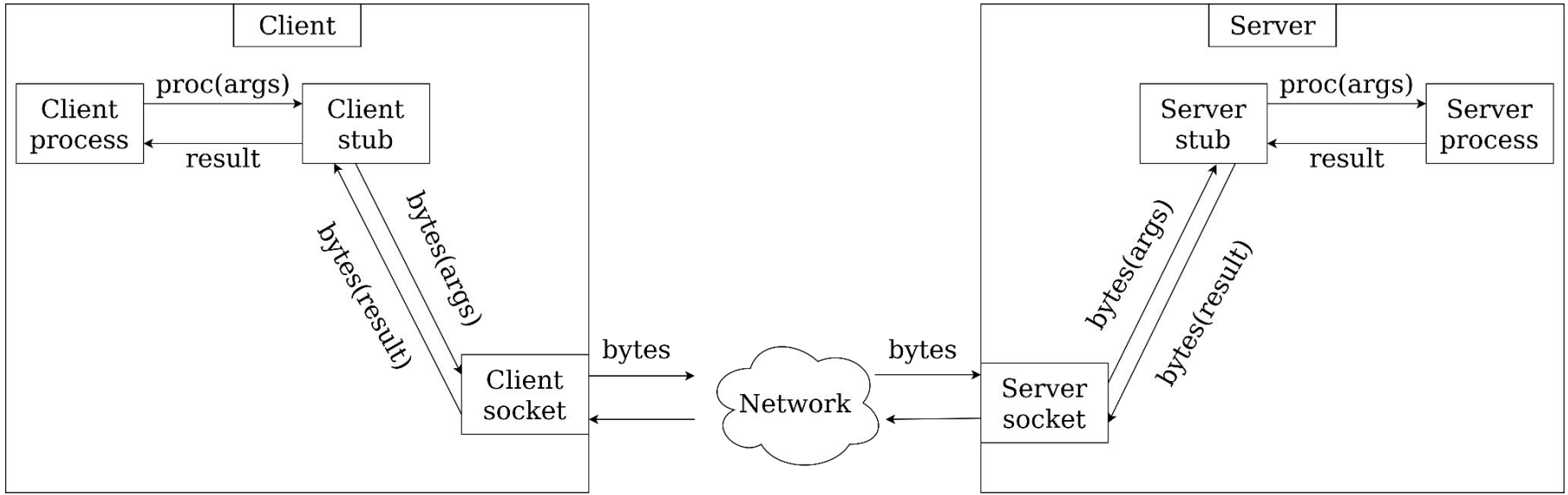


RPC: Переиспользование канала

- Делаем несколько RPC-вызовов
  - `x := RPC.GetValue(“x”)`
  - `y := RPC.GetValue(“y”)`
  - `RPC.PutValue(“x + y”, x + y)`
- Использовать для каждого вызова отдельный сокет дорого
  - Создание сокета это операция в ядре ОС

Архитектура типичного RPC-фреймворка

- Заглушки решают вышеупомянутые проблемы
- Пользователь реализует только логику операций



gRPC: описание интерфейса

- Параметр - произвольное Protobuf-сообщение
  - Параметр ровно один
- Результат - произвольное Protobuf-сообщение
- Произвольное число процедур у сервиса

```
service RPCService {
  rpc ProcessRequest(Request) returns (Response);
}

message Request {
  uint64 id = 1;
  string key = 2;
  repeated string value = 3;
}

message Response {
  oneof response {
    string message = 1;
    uint32 error_code = 2;
  }
}
```

gRPC: Client & Server Stubs

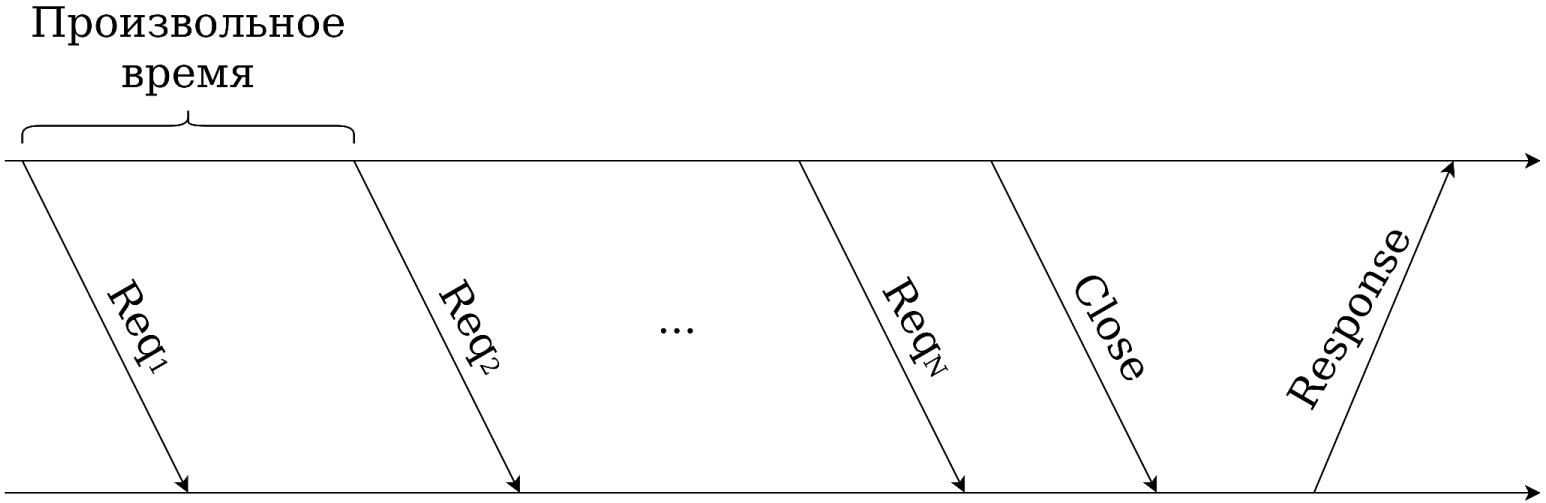
- gRPC генерирует код Client Stub и Server Stub
- Под капотом использует TCP

```go
func (c *greeterClient) SayHello(ctx context.Context, in *HelloRequest, opts ...grpc.CallOption) (*HelloResponse, error) {
    out := new(HelloResponse)
    err := c.cc.Invoke(ctx, "/_1_hello.Greeter/SayHello", in, out, opts...)
    if err != nil {
        return nil, err
    }
    return out, nil
}
```

```go
func _Greeter_SayHello_Handler(srv interface{}, ctx context.Context, dec func(interface{}) error, interceptor grpc.UnaryServerInterceptor) (interface{}, error) {
    in := new(HelloRequest)
    if err := dec(in); err != nil { return nil, err }
    if interceptor == nil { return srv.(GreeterServer).SayHello(ctx, in) }
    info := &grpc.UnaryServerInfo{
        Server:     srv,
        FullMethod: "/_1_hello.Greeter/SayHello",
    }
    handler := func(ctx context.Context, req interface{}) (interface{}, error) {
        return srv.(GreeterServer).SayHello(ctx, req.(*HelloRequest))
    }
    return interceptor(ctx, in, info, handler)
}
```

gRPC: Client streaming

- Клиент отдаёт данные по частям
- Между соседними частями проходит произвольное время
- В конце сервер возвращает ответ
- Реализация: TCP + Фрейминг + Маркер конца



gRPC: Client streaming

```
service StreamConsumer {
  rpc ConsumeStream (stream StreamEntity) returns (Response);
}
```

- Объявляем как обычное RPC
  - Но помечаем параметр как `stream`
- В том же сервисе может быть объявлено произвольное количество других RPC
  - Произвольного типа

gRPC: Client streaming

```
1 fun Procedure(server_stream):
2     result := ...
3     while true:
4         msg, eof := server_stream.Recv()
5         if eof:
6             return result
7         result ← combine(result, msg)
```

- Сервер принимает сообщения с помощью `data, eof := stream.Recv()`
  - `eof = true` когда клиент закрывает поток
  - В этот момент надо вернуть клиенту ответ

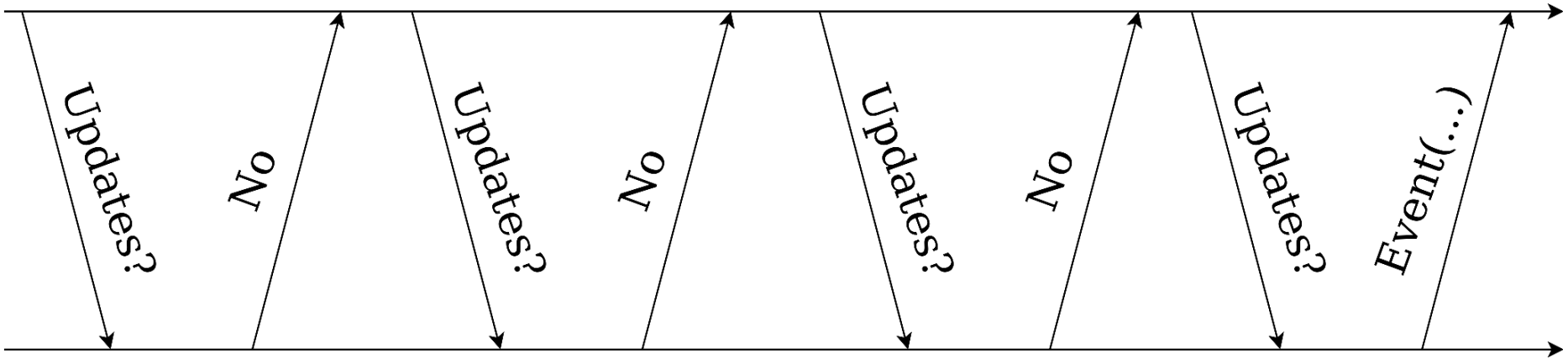
gRPC: Client streaming

```
1 client_stream := client.Procedure()
2 while has_data:
3     msg := get_data()
4     client_stream.Send(msg)
5 result := client_stream.SendAndClose()
```

- Клиент создаёт поток с помощью `stream := client.Procedure()`
- Посылает сообщения с помощью `stream.Send(x)`
- Закрывает поток и получает ответ с помощью `result := stream.CloseAndRecv()`

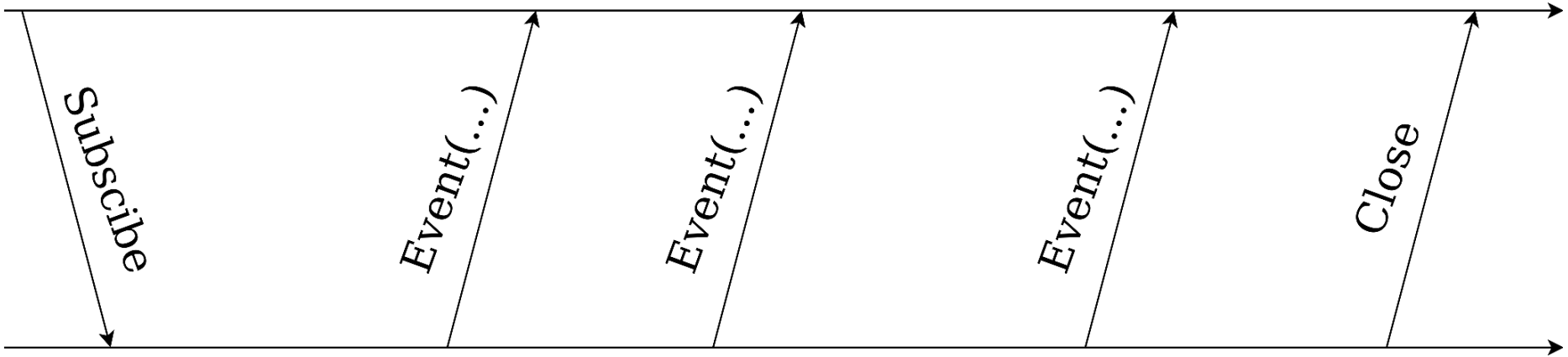
gRPC: Server streaming

- Решаем проблему проверки обновления интересующих нас данных
- Polling
  - Много бессмысленных сообщений
  - Отъедаем RPS сервера и Bandwidth сети на бесполезные действия



gRPC: Server streaming

- Подписываемся на интересующие нас события
- Получаем поток этих событий с сервера
  - Только тогда, когда события реально происходят
- Реализация: TCP + Фрейминг + Маркер конца



gRPC: Server streaming

```
service SubscriptionService {
  rpc Subscribe (Subscription) returns (stream Event);
}
```

- Объявляем как обычное RPC
  - Но помечаем возвращаемое значение как `stream`
- В том же сервисе может быть объявлено произвольное количество других RPC
  - Произвольного типа

gRPC: Server streaming

```
1 fun Procedure(recv, server_stream):
2     state := init_state(req)
3     while has_data(state):
4         msg := get_data(state)
5         state ← update_state(state, msg)
6         server_stream.Send(msg)
```

- Сервер отправляет данные с помощью `stream.Send(x)`
  - Возвращается из функции-обработчика когда данные кончились

gRPC: Server streaming

```
1 client_stream := client.Procedure(args)
2 while true:
3     msg, eof := client_stream.Recv()
4     if eof:
5         break
6     consume(msg)
```

- Клиент создаёт поток с помощью
  `stream := client.Procedure(args)`
- Получает данные с помощью
  `x, eof := stream.Recv()`
  - `eof = true` когда сервер закрывает поток

gRPC: Bidirectional streaming

```
service MessengerService {
  rpc StartMessaging (stream Message) returns (stream Message);
}
```

- Устанавливаем соединение с помощью
  `stream := client.Procedure()`
- Читаем данные с помощью
  `data, eof := stream.Recv()`
  - `eof = true` когда другая сторона перестаёт писать
- Отправляем данные с помощью
  `eof := stream.Send(x)`
  - `eof = true` когда другая сторона перестаёт читать
- Перестаём писать с помощью `stream.CloseSend()`
- Перестаём читать выходя из обработчика

Прочие RPC-фреймворки

- Java Remote Method Invocation
- .NET Remoting
- Sun RPC
  - Используется в Network File System
- Текстовые протоколы поверх HTTP
  - JSON RPC
  - XML RPC
  - SOAP

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org
/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
     <getProductDetails xmlns="http://warehouse.example.com/ws">
       <productID>12345</productID>
     </getProductDetails>
   </soap:Body>
</soap:Envelope>
```

Что почитать

- gRPC & Go Basic Tutorial
  - grpc.io/docs/languages/go/basics
- Road to gRPC
  - blog.cloudflare.com/road-to-grpc
- *Сергей Камардин.* Миллион WebSocket и Go
  - habr.com/ru/company/vk/blog/331784

Thanks for your attention