

## 4 Задание 4. Разработка приложения на основе gRPC

<b>4</b>	<b>Задание 4. Разработка приложения на основе gRPC.....</b>	<b>1</b>
<b>4.1</b>	<b>Критерии оценивания.....</b>	<b>1</b>
<b>4.2</b>	<b>Методические указания .....</b>	<b>2</b>
<b>4.3</b>	<b>Примеры реализации .....</b>	<b>3</b>
4.3.1	Подготовка к разработке .....	3
4.3.2	Структура проекта.....	3
4.3.3	.go.mod -файл.....	4
4.3.4	.proto-файл .....	4
4.3.5	Сервер.....	5
4.3.6	Клиент.....	7
4.3.7	Запуск и тестирование приложения .....	8
<b>4.4</b>	<b>Задание на самостоятельную работу .....</b>	<b>8</b>
<b>4.5</b>	<b>Ссылки .....</b>	<b>8</b>

**Цель:** на языке высокого уровня (Java, C#, Python и др. – на выбор обучающегося) реализовать gRPC веб-сервис и клиент для него, обеспечивающий работу движка игры «SOA-мафия».

### 4.1 Критерии оценивания

№	Задача	Баллы
1.	Реализовать базовое клиент-серверное приложение «Привет Мафия» на основе технологии gRPC. <i>Клиент обеспечивает:</i> <ul style="list-style-type: none"><li>1) Установку имени пользователя</li><li>2) Подключение к серверу по сетевому имени/адресу</li><li>3) Отображение списка подключившихся игроков</li></ul> <i>Сервер обеспечивает:</i> <ul style="list-style-type: none"><li>1) Подключение игроков</li><li>2) Рассылку уведомлений о подключившихся/отключившихся игроках</li></ul>	4
2.	Реализовать базовое приложение «Боты мафии». Оно реализует все возможности приложения «Привет Мафия», а также: <i>Клиент обеспечивает:</i> <ul style="list-style-type: none"><li>1) Автоматический вход в сессию игры, когда набирается достаточное число игроков</li><li>2) Отображение состояния игрока и действий, происходящих в игре</li><li>3) Случайный выбор действий среди возможных на каждом этапе игры</li></ul>	6

	<p><i>Сервер обеспечивает:</i></p> <ol style="list-style-type: none"> <li>1) Создание одной сессии игры при подключении достаточного количества игроков</li> <li>2) Назначение игрокам случайных ролей в соответствии с требованиями</li> <li>3) Получение от игроков выбранных действий, их выполнение и изменение состояния игроков</li> <li>4) Учет статуса игры и завершение игры при выигрыше мирных жителей или мафии</li> </ol>	
3.	<p>Модифицировать клиент и сервер таким образом, чтобы они обеспечивали:</p> <ol style="list-style-type: none"> <li>1. Возможность общения игроков внутри сессии, с учетом состояния игры: днем все игроки могут свободно общаться, ночью могут общаться только игроки мафии между собой, «духи» отключаются от общения до новой сессии игры. Для реализации общения можно использовать сервис чата, разработанный в предыдущей работе.</li> <li>2. Одновременное ведение нескольких сеансов игры при подключении достаточного количества игроков.</li> </ol>	5

## 4.2 Методические указания

Объединяющим мотивом для следующей серии практических заданий будет создание компонентов игры «SOA-Мафия». Для реализации отдельных аспектов этой игры необходимо будет реализовать следующие сервисы (и, соответственно, клиентов к ним):

- Сервис голосового чата, обеспечивающий обмен сообщениями между участниками игры (на технологии сокетов);
- Сервис, реализующий движок игры (на технологии RPC);
- Сервис для организации работы с информацией о сессиях игры, статистикой игроков и достижениях (с возможностью доступа по REST и GraphQL).

Игра должна представлять из себя сетевой вариант игры Мафия. Взаимодействие между клиентом и сервером обеспечивается на технологии gRPC.

Когда достаточное количество клиентов подключается к серверу, сервер принимает решение о ролях участников и уведомляет участников о них. Логика распределения ролей определяется разработчиком, в минимальном варианте - игра на 4 игрока должна иметь одного игрока с ролью “мафия” и одного игрока с ролью “комиссар”.

Общий процесс игры:

- Игра начинается в игровой день
- Пока идет игровой день, все участники могут общаться в общем чате, либо отправлять серверу определенные команды. Можно использовать сервис чата, разработанный в рамках предыдущей лабораторной работы.
- Игрок может выполнить команду “завершить день”. Когда все участники отправляют эту команду серверу, игровой день завершается.

- Игрок может выполнить команду “казнить игрока” в любой день, кроме первого. Игрок, за которого проголосовало большинство, переводится в состояние «духа» (он может наблюдать за дальнейшим ходом игры, но не может участвовать в обсуждении и влиять на ход игры).
- Когда игровой день заканчивается, наступает ночь и сервер уведомляет игрока(ов) с ролью “мафия” и игрока(ов) с ролью “комиссар” о том, что им требуется принять решение.
  - Мафия принимает решение о том, кто из игроков должен быть “убит” (в этом случае игрок переводится в состояние «духа»).
  - Комиссар(ы) выбирают игрока, которого нужно проверить. Если проверяемый является мафией, то комиссар на следующий день может отправить команду “опубликовать данные”, при этом сервер публикует информацию о том, кто является мафией.
- Игровой день начинается снова.

Игра считается завершенной, когда количество игроков с ролью “мафия” становится равно количеству игроков без роли (мирных жителей) либо когда количество игроков с ролью “мафия” становится равно нулю.

Детали реализации, организации коммуникации и голосования, обсуждения и всё остальное – остается на откуп разработчику.

## 4.3 Примеры реализации

### 4.3.1 Подготовка к разработке

Для работы с проектом потребуется предварительная установка и настройка следующих компонентов:

1. Язык Go (версия 1.15) <https://golang.org/>
2. Утилита protoc для обработки .proto файлов:  
<https://github.com/protocolbuffers/protobuf/releases/tag/v3.14.0>
3. Набор модулей для языка Go для работы с gRPC и Protobuf:  
<https://grpc.io/docs/languages/go/quickstart/>
4. Возможно потребуется их обновление в ручном режиме командами:
 

```
go get -u github.com/golang/protobuf/proto
go get -u github.com/golang/protobuf/protoc-gen-go
go get -u google.golang.org/grpc
```

### 4.3.2 Структура проекта

Наш проект будет состоять из трех компонентов:

1. Протокол, автоматически генерируемый из .proto – файла
2. Сервер
3. Клиент

Создадим следующую структуру проекта:

```
grpc-go-reverse
├── client
│   └── main.go
├── go.mod
├── go.sum
├── pkg
│   ├── proto
│   │   └── reverse
│   │       └── reverse.proto
└── server
    └── main.go
```

#### 4.3.3 .go.mod -файл

Для работы с системой модулей в проекте, необходимо в папке разместить файл `go.mod`. Для того, чтобы мы могли пользоваться модулями, созданными нашим генератором, можно отредактировать `go.mod`, воспользовавшись директивой `replace`, добавив в конце файла:

```
replace github.com/damage/grpc-queue/reverse => ../pkg/proto/reverse
```

#### 4.3.4 .proto-файл

**.proto**-файл описывает, какие операции наш сервис будет осуществлять и какими данными он при этом будет обмениваться. Создаем в проекте папку **proto**, а в ней — файл **reverse.proto**

```
syntax = "proto3";

package reverse;

service Reverse {
    rpc Do(Request) returns (Response) {}
}

message Request {
    string message = 1;
}

message Response {
    string message = 1;
}
```

```
}
```

Функция, которая вызывается удаленно на сервере и возвращает данные клиенту, помечается как ***rpc***. Структуры данных, служащие для обмена информацией между взаимодействующими узлами, помечаются как ***message***. Каждому полю сообщения необходимо присвоить порядковый номер. В данном случае наша функция принимает от клиента сообщения типа ***Request*** и возвращает сообщения типа ***Response***.

Как только мы создали **.proto**-файл, необходимо получить **.go**-файл нашего сервиса. Для этого нужно выполнить следующую консольную команду в папке **proto**:

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

Разумеется, сначала вам нужно выполнить [сборку gRPC](#).

Выполнение вышеприведенной команды создаст набор **.go**-файлов:

reverse.pb.go – код для сериализации/десериализации сообщений  
reverse\_grpc.pb.go – заглушки для клиента и сервера

#### 4.3.5 Сервер

Подготовим код сервера

Рассмотрим ключевые элементы, данного кода:

- 1) Блок импорта: импортируем все необходимые библиотеки, плюс ссылаемся на наш protobuf-модуль

```
import (  
    "context"  
    "fmt"  
    "log"  
    "net"  
  
    "github.com/damage/grpc-queue/reverse"  
    "google.golang.org/grpc"  
)
```

- 
- 2) Описание структуры, реализующей интерфейс нашего сервера ReverseServerInterface (в Go реализация интерфейса происходит неявно путем имплементации методов интерфейса). Также, в структуру произведено внедрение "UnimplementedReverseServer", для того чтобы в процессе реализации методов сервера система автодополнения могла подсказать, какие методы еще не реализованы.

После описания структуры, мы, собственно, реализуем единственный метод, который должен реализовать наш сервер – DoReverse.

```
type server struct {
    reverse.UnimplementedReverseServer
}

func (s *server) DoReverse(ctx context.Context, request *reverse.Request) (*reverse.Response, error) {
    log.Println(fmt.Sprintf("Request: %s", request.GetMessage()))
    return &reverse.Response{Message: fmt.Sprintf("Reversed string %s", stringReverse(request.GetMessage()))}, nil
}
```

- 3) Основной метод, запускающий tcp-сервер на порту 9000 и создающий gRPC сервер. Также, функция переворота строки, ради которой всё и затевалось.

```
func main() {

    lis, err := net.Listen("tcp", ":9000")
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }

    srv := grpc.NewServer()
    reverse.RegisterReverseServer(srv, &server{})

    log.Fatalln(srv.Serve(lis))
}

func stringReverse(str string) (result string) {
    for _, v := range str {
        result = string(v) + result
    }
    return
}
```

#### 4.3.6 Клиент

Аналогично, рассмотрим код клиента

- 1) Блок импорта: импортируем все необходимые библиотеки, плюс ссылаемся на наш protobuf-модуль

```
import (  
    "context"  
    "log"  
    "time"  
  
    "github.com/domage/grpc-queue/reverse"  
    "google.golang.org/grpc"  
)
```

- 2) Основной метод, в котором происходит подключение к tcp-серверу по порту 9000, создание клиента на основе proto-файла и вызов удаленной функции.

```
func main() {  
    log.Println("Client running ...")  
  
    conn, err := grpc.Dial(":9000", grpc.WithInsecure(), grpc.WithBlock()  
)  
    if err != nil {  
        log.Fatalln(err)  
    }  
    defer conn.Close()  
  
    client := reverse.NewReverseClient(conn)  
  
    request := &reverse.Request{Message: "This is a test"}  
  
    ctx, cancel := context.WithTimeout(context.Background(), time.Second  
)  
    defer cancel()  
  
    response, err := client.DoReverse(ctx, request)  
    if err != nil {  
        log.Fatalln(err)  
    }  
}
```

```
log.Println("Response:", response.GetMessage())  
}
```

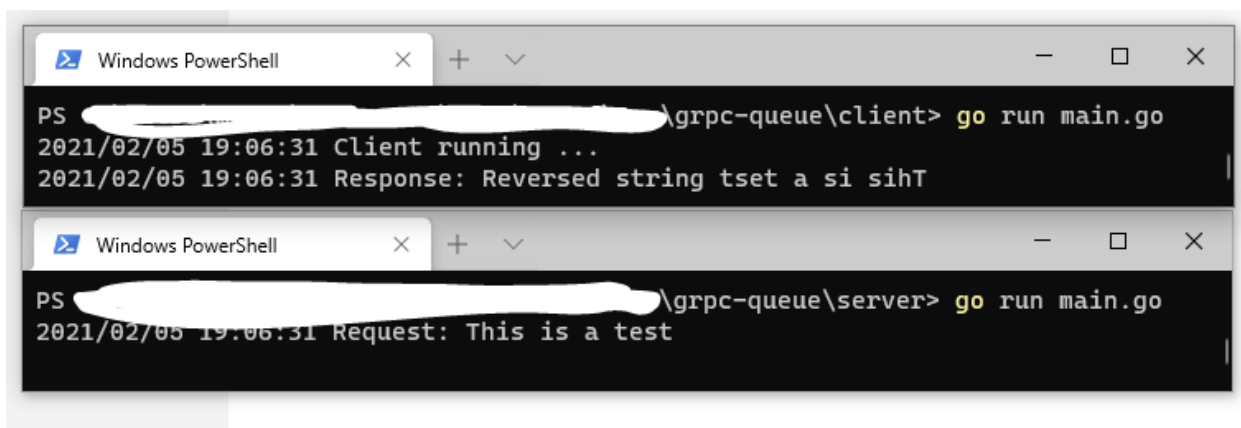
#### 4.3.7 Запуск и тестирование приложения

Теперь мы можем запустить наше приложение.

Запускаем сервер и клиент командой

```
$ go run main.go
```

В результате успешного запуска получаем результат:



The screenshot shows two overlapping Windows PowerShell windows. The top window, titled 'Windows PowerShell', shows the command `go run main.go` being executed in a directory `\grpc-queue\client>`. The output shows the client running and receiving a response: `2021/02/05 19:06:31 Client running ...` and `2021/02/05 19:06:31 Response: Reversed string tset a si sihT`. The bottom window, also titled 'Windows PowerShell', shows the command `go run main.go` being executed in a directory `\grpc-queue\server>`. The output shows the server running and receiving a request: `2021/02/05 19:06:31 Request: This is a test`.

#### 4.4 Задание на самостоятельную работу

На основе представленных примеров вам предлагается самостоятельно реализовать gRPC веб-сервис, обеспечивающий решение задач 1-3.

#### 4.5 Ссылки

Для дальнейшего самостоятельного изучения данной темы можно воспользоваться следующими ресурсами:

1. Создание простейшего gRPC сервиса на go: <https://grpc.io/docs/languages/go/basics/>
2. Еще один актуальный пример: <http://www.inanzzz.com/index.php/post/fczz/creating-a-simple-grpc-client-and-server-application-with-golang>
3. Создание gRPC сервиса на Node.js  
<https://codelabs.developers.google.com/codelabs/cloud-grpc-ru/index.html?index=..%2F..lang-ru#0>