Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Московский государственный технический университет
имени Н.Э. Баумана»
(МГТУ им. Н.Э. Баумана)

Факультет: Информатика и системы управления
Кафедра: Теоретическая информатика и компьютерные технологии

# Лабораторная работа №1
# «Простейший протокол прикладного уровня»
# по курсу: «Компьютерные сети»

Выполнил:
Студент группы ИУ9-32Б
Гнатенко Т. А.

Проверил:
Посевин Д. П.

Москва, 2022

# Цели

Целью работы является знакомство с принципами разработки протоколов прикладного уровня и их реализацией на языке Go.

# Задачи

1. Разработать вариант протокола из таблиц 1–3. Протокол должен базироваться на текстовых сообщениях в формате JSON. Результатом разработки протокола должен быть набор типов языка Go, представляющих сообщения, и документация к ним в виде комментариев в исходном тексте.
2. Написать на языке Go клиент и сервер, взаимодействующие по разработанному протоколу. Основные требования к клиенту и серверу:
3. полная проверка данных, получаемых из сети (необходимо учитывать, что сообщения могут приходить в неправильном формате и в неправильном порядке, а также могут содержать неправильные данные);
4. устойчивость к обрыву соединения;
5. возможность одновременного подключения нескольких клиентов к одному серверу;
6. сервер должен вести подробный лог всех ошибок, а также других важных событий (установка и завершение соединения с клиентом, приём и передача сообщений, и т.п.).

Протокол поиска подпоследовательности целых чисел с максимальной суммой.

# Решение

## Исходный код

**client.go**

```
package main

import (
    "encoding/json"
    "flag"
    "fmt"
    "lab1/src/proto"
    "net"
)

// interact - функция, содержащая цикл взаимодействия с
 ↪  сервером.
func interact(conn *net.TCPConn) {
    defer conn.Close()
    encoder, decoder := json.NewEncoder(conn),
 ↪  json.NewDecoder(conn)
    for {
        // Чтение команды из стандартного потока ввода
        fmt.Printf("command = ")
        var command string
        fmt.Scan(&command)

        // Отправка запроса.
        switch command {
        case "quit":
            sendRequest(encoder, "quit", nil)
            return
        case "find":
            var task proto.Task
            fmt.Printf("length sequence = ")
            fmt.Scan(&task.Length)
            fmt.Printf("sequence: ")
            var a int
            for i := 0; i < task.Length; i++ {
                fmt.Scan(&a)
```

```
                task.Sequence = append(task.Sequence, a)
            }
            sendRequest(encoder, "find", &task)
        default:
            fmt.Printf("error: unknown command\n")
            continue
        }

        // Получение ответа.
        var resp proto.Response
        if err := decoder.Decode(&resp); err != nil {
            fmt.Printf("error: %v\n", err)
            break
        }

        // Вывод ответа в стандартный поток вывода.
        switch resp.Status {
        case "ok":
            fmt.Printf("ok\n")
        case "failed":
            if resp.Data == nil {
                fmt.Printf("error: data field is absent in
↪  response\n")
            } else {
                var errorMsg string
                if err := json.Unmarshal(*resp.Data,
                 ↪  &errorMsg); err != nil {
                    fmt.Printf("error: malformed data field
↪  in response\n")
                } else {
                    fmt.Printf("failed: %s\n", errorMsg)
                }
            }
        case "result":
            if resp.Data == nil {
                fmt.Printf("error: data field is absent in
↪  response\n")
            } else {
                var subsequence []interface{}
                if err := json.Unmarshal(*resp.Data,
                 ↪  &subsequence); err != nil {
```

```
                    fmt.Printf("error: malformed data field
 ↪  in response\n")
                } else {
                    fmt.Print("subsequence with the max sum:
 ↪  ")
                    for _, v := range subsequence {
                        fmt.Printf("%v ", v)
                    }
                    fmt.Println()
                }
            }
        default:
            fmt.Printf("error: server reports unknown status
 ↪  %q\n", resp.Status)
        }
    }
}

// sendRequest - вспомогательная функция для передачи
 ↪  запроса с указанной командой
// и данными. Данные могут быть пустыми (data == nil).
func sendRequest(encoder *json.Encoder, command string, data
 ↪  interface{}) {
    var raw json.RawMessage
    raw, _ = json.Marshal(data)
    encoder.Encode(&proto.Request{Command: command, Data:
 ↪  &raw})
}

func main() {
    // Работа с командной строкой, в которой может
     ↪  указываться необязательный ключ -addr.
    var addrStr string
    flag.StringVar(&addrStr, "addr", "127.0.0.1:6000",
 ↪  "specify ip address and port")
    flag.Parse()

    // Разбор адреса, установка соединения с сервером и
    // запуск цикла взаимодействия с сервером.
    if addr, err := net.ResolveTCPAddr("tcp", addrStr); err
     ↪  != nil {
```

```
        fmt.Printf("error: %v\n", err)
    } else if conn, err := net.DialTCP("tcp", nil, addr);
    ↪ err != nil {
        fmt.Printf("error: %v\n", err)
    } else {
        interact(conn)
    }
}
```

**proto.go**

```
package proto

import "encoding/json"

// Request -- запрос клиента к серверу.
type Request struct {
    // Поле Command может принимать три значения:
    // * "quit" - прощание с сервером (после этого сервер
    ↪  рвёт соединение);
    // * "calculate" - передача новой задачи на сервер;
    Command string `json:"command"`

    Data *json.RawMessage `json:"data"`
}

// Response -- ответ сервера клиенту.
type Response struct {
    // Поле Status может принимать три значения:
    // * "ok" - успешное выполнение команды "quit";
    // * "failed" - в процессе выполнения команды произошла
    ↪  ошибка;
    // * "result" - максимальная высота вычислена.
    Status string `json:"status"`

    // Если Status == "failed", то в поле Data находится
    ↪  сообщение об ошибке.
    // Если Status == "result", в поле Data должна лежать
    ↪  подпоследовательность
    // В противном случае, поле Data пустое.
```

```
    Data *json.RawMessage `json:"data"`
}

// Task -- условие задачи для вычисления сервером
type Task struct {
    // Длина подпоследовательности
    Length int `json:"length"`

    // Последовательность чисел
    Sequence []int `json:"sequence"`
}
```

**server.go**

```
package main

import (
    "encoding/json"
    "flag"
    "fmt"
    "lab1/src/proto"
    "net"

    log "github.com/mgutz/logxi/v1"
)

// Client - состояние клиента.
type Client struct {
    logger log.Logger    // Объект для печати логов
    conn   *net.TCPConn  // Объект TCP-соединения
    enc    *json.Encoder // Объект для кодирования и
 ↪  отправки сообщений
    count  int64         // Количество полученных от клиента
 ↪  задач
}

// NewClient - конструктор клиента, принимает в качестве
 ↪  параметра
// объект TCP-соединения.
func NewClient(conn *net.TCPConn) *Client {
```

```
    return &Client{
        logger: log.New(fmt.Sprintf("client %s",
 ↪  conn.RemoteAddr().String())),
        conn:   conn,
        enc:    json.NewEncoder(conn),
        count:  0,
    }
}

// serve - метод, в котором реализован цикл взаимодействия с
 ↪  клиентом.
// Подразумевается, что метод serve будет вызываться в
 ↪  отдельной go-программе.
func (client *Client) serve() {
    defer client.conn.Close()
    decoder := json.NewDecoder(client.conn)
    for {
        var req proto.Request
        if err := decoder.Decode(&req); err != nil {
            client.logger.Error("cannot decode message",
 ↪  "reason", err)
            break
        } else {
            client.logger.Info("received command",
 ↪  "command", req.Command)
            if client.handleRequest(&req) {
                client.logger.Info("shutting down
 ↪  connection")
                break
            }
        }
    }
}

// handleRequest - метод обработки запроса от клиента. Он
 ↪  возвращает true,
// если клиент передал команду "quit" и хочет завершить
 ↪  общение.
func (client *Client) handleRequest(req *proto.Request) bool
 ↪  {
```

```
    switch req.Command {
    case "quit":
        client.respond("ok", nil)
        return true
    case "find":
        errorMsg := ""
        var x []int
        if req.Data == nil {
            errorMsg = "data field is absent"
        } else {
            var task proto.Task
            if err := json.Unmarshal(*req.Data, &task); err
             ↪  != nil {
                errorMsg = "malformed data field"
            } else {
                var sum int
                ans := 0
                ans_l := 0
                ans_i := 0
                min_pos := -1
                sequence := task.Sequence
                n := len(sequence)
                for i := 0; i < n; i++ {
                    sum += sequence[i]

                    if sum > ans {
                        ans = sum
                        ans_l = min_pos + 1
                        ans_i = i
                    }

                    if sum < 0 {
                        sum = 0
                        min_pos = i
                    }
                }
                for i := ans_l; i <= ans_i; i++ {
                    x = append(x, sequence[i])
                }
                client.logger.Info("performing
↪   calculations", "value", x)
```

```
                client.count++
            }
        }
        if errorMsg == "" {
            client.respond("result", &x)
        } else {
            client.logger.Error("calculation failed",
 ↪  "reason", errorMsg)
            client.respond("failed", errorMsg)
        }
    default:
        client.logger.Error("unknown command")
        client.respond("failed", "unknown command")
    }
    return false
}

// respond - вспомогательный метод для передачи ответа с
 ↪  указанным статусом
// и данными. Данные могут быть пустыми (data == nil).
func (client *Client) respond(status string, data
 ↪  interface{}) {
    var raw json.RawMessage
    raw, _ = json.Marshal(data)
    client.enc.Encode(&proto.Response{Status: status, Data:
 ↪  &raw})
}

func main() {
    // Работа с командной строкой, в которой может
     ↪  указываться необязательный ключ -addr.
    var addrStr string
    flag.StringVar(&addrStr, "addr", "127.0.0.1:6000",
 ↪  "specify ip address and port")
    flag.Parse()

    // Разбор адреса, строковое представление которого
     ↪  находится в переменной addrStr.
    if addr, err := net.ResolveTCPAddr("tcp", addrStr); err
     ↪  != nil {
```

```
        log.Error("address resolution failed", "address",
    ↪  addrStr)
    } else {
        log.Info("resolved TCP address", "address",
    ↪  addr.String())

        // Инициация слушания сети на заданном адресе.
        if listener, err := net.ListenTCP("tcp", addr); err
         ↪  != nil {
            log.Error("listening failed", "reason", err)
        } else {
            // Цикл приёма входящих соединений.
            for {
                if conn, err := listener.AcceptTCP(); err !=
                 ↪  nil {
                    log.Error("cannot accept connection",
    ↪  "reason", err)
                } else {
                    log.Info("accepted connection",
    ↪  "address", conn.RemoteAddr().String())

                    // Запуск go-программы для обслуживания
                     ↪  клиентов.
                    go NewClient(conn).serve()
                }
            }
        }
    }
}
```

## Вывод

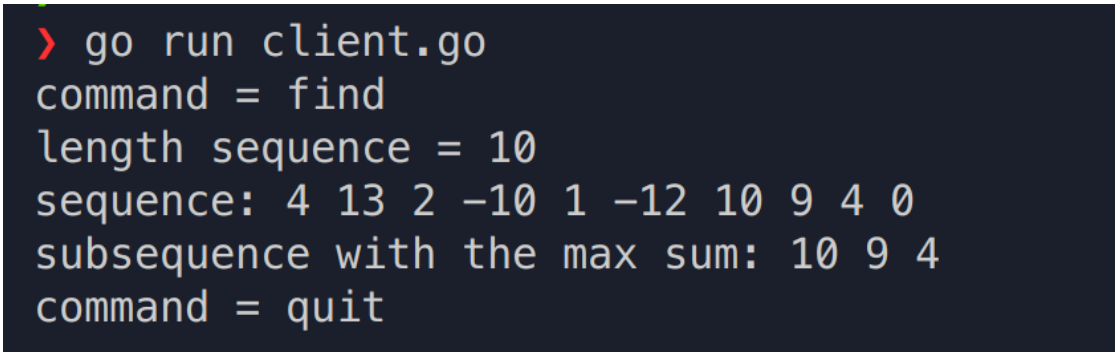


Рис. 1: Отправка запроса и получение ответа