**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**
**TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN**

# BÁO CÁO LAB 01
## XÂY DỰNG MÔ HÌNH CLIENT-SERVER CHO MÃ NGUỒN KEY-VALUE STORE


**THÀNH VIÊN NHÓM:** Nguyễn Thiên Phúc

**GVHD:** TS. Ngô Huy Biên

ThS. Ngô Ngọc Đăng Khoa


**NĂM HỌC 2024 – 2025**

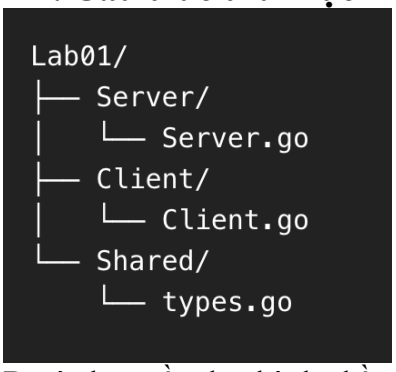
# MỤC LỤC

# I. Thông tin sinh viên

- 20127681 – Nguyễn Thiên Phúc

# II. Cấu trúc dự án

## 1. Cấu trúc thư mục mã nguồn của dự án



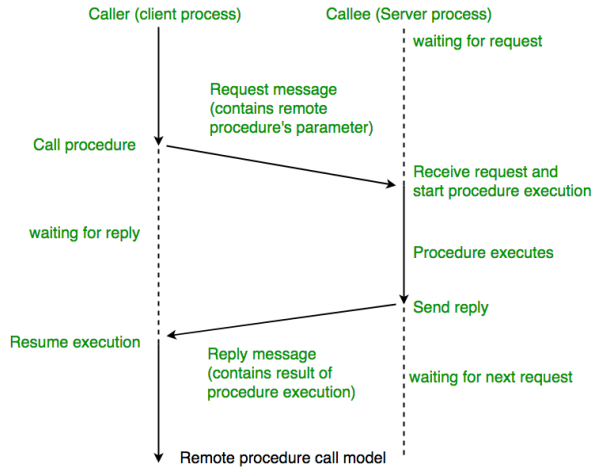

Dự án bao gồm ba thành phần chính:

- Client (Client.go): Quản lý việc gửi yêu cầu Put (ghi dữ liệu) và Get (truy vấn dữ liệu) đến server.
- Server (Server.go): Quản lý dữ liệu khóa-giá trị và xử lý các yêu cầu từ client.
- Shared Types (types.go): Định nghĩa các cấu trúc dữ liệu và mã lỗi dùng chung giữa client và server.

## 2. Các thư viện sử dụng trong dự án

| Tên thư viện | Mô tả |
|---|---|
| net/rpc | Thư viện cung cấp các hàm để thực hiện Remote Procedure Call (RPC) trong Go. Đây là nền tảng cho việc giao tiếp giữa client và server trong dự án. |
| net | Thư viện hỗ trợ các thao tác liên quan đến mạng, bao gồm việc lắng nghe kết nối TCP, xử lý địa chỉ mạng, và giao tiếp qua socket. |
| sync | Thư viện cung cấp các cấu trúc đồng bộ hóa như Mutex, WaitGroup và các công cụ khác để quản lý các hoạt động đồng thời. |
| fmt | Thư viện tiêu chuẩn của Go để định dạng và in thông tin ra màn hình. |
| log | Thư viện tiêu chuẩn để ghi log thông tin và báo lỗi. |
| strconv | Thư viện tiêu chuẩn để chuyển đổi giữa các kiểu dữ liệu cơ bản (string, int, float, ...) |

### 3. Luồng hoạt động của hệ thống




Hình 1: Sơ đồ hoạt động của hệ thống Server-Client bằng RPC framework
(Nguồn: https://www.geeksforgeeks.org/remote-procedure-call-rpc-in-operating-system/)

**Server:** Lắng nghe yêu cầu → Thực thi thủ tục → Gửi lại kết quả → Tiếp tục chờ yêu cầu tiếp theo.
**Client:** Gọi hàm → Chờ phản hồi → Nhận kết quả → Tiếp tục gửi yêu cầu.

## III. Mô tả các thành phần của dự án

### 1. File types.go

#### a. Chức năng

- Định nghĩa các cấu trúc dữ liệu dùng chung giữa client và server.
- Xác định mã lỗi chuẩn.

#### b. Cấu trúc của file

- Hai struct là PutArgs và PutReply: Định nghĩa tham số và phản hồi cho phương thức

```
type PutArgs struct {  2 usages  new *
    Key   string
    Value string
}

Implement interface
type PutReply struct {  2 usages  new *
    Err Err
}
```

- Hai struct là GetArgs và GetReply: Định nghĩa tham số và phản hồi cho phương thức Get

```
Implement interface
type GetArgs struct {  2 usages  new *
    Key string
}
Implement interface
type GetReply struct {  2 usages  new *
    Err   Err
    Value string
}
```

- Tiếp đến là hằng số lỗi, dùng để xác định mã lỗi như “OK” hoặc “ErrNoKey”

```
const (
    OK       = "OK"  2 usages  new *
    ErrNoKey = "ErrNoKey"  1 usage  new *
)
```

## 2. File Server.go

### a. Chức năng chính

- Quản lý dữ liệu key-value bằng cấu trúc dữ liệu map của Go.
- Xử lý đồng thời nhiều kết nối client với RPC.

### b. Cấu trúc của file

```
type KV struct {  3 usages  new *
    mu   sync.Mutex
    data map[string]string
}
```

- Đầu tiên, định nghĩa một cấu trúc để lưu trữ các cặp key-value

- **Hàm Put**

```
func (kv *KV) Put(args *kvs.PutArgs, reply *kvs.PutReply) error {  no usages  new *
    kv.mu.Lock()
    defer kv.mu.Unlock()

    kv.data[args.Key] = args.Value
    reply.Err = kvs.OK
    return nil
}
```

  - **kv *KV:** Hàm là một phương thức của cấu trúc KV, cấu trúc này chứa dữ liệu khóa-giá trị và một Mutex để đồng bộ hóa.
  - **args *kvs.PutArgs**: Tham số đầu vào, chứa:

- **args.Key:** Khóa cần lưu.
- **args.Value:** Giá trị cần lưu cho khóa đó.
- **reply *kvs.PutReply:** Phản hồi trả về cho client, chứa thông tin về trạng thái thực hiện (Err).
- **error:** Hàm trả về lỗi nếu có lỗi trong quá trình lưu.

- Đầu tiên, hàm thực hiện gọi kv.mu.Lock() để khóa dữ liệu trước khi thực hiện bất kỳ thao tác ghi nào trên map kv.data.
- Sử dụng defer kv.mu.Unlock() để đảm bảo rằng Mutex luôn được mở khóa sau khi kết thúc hàm, ngay cả khi có lỗi xảy ra.
- Sử dụng args.Key làm khóa và args.Value làm giá trị để lưu vào bản đồ kv.data
- Trả về trạng thái thành công (kvs.OK) thông qua tham chiếu reply.Err
- Hàm không có lỗi nên trả về nil

- **Hàm getAvailablePort**

```
func getAvailablePort(preferredPort string) string {  1 usage   new *
    listener, err := net.Listen( network: "tcp", preferredPort)
    if err != nil {
        fmt.Printf( format: "Port %s is already in use. Choosing a different port...\n", preferredPort)
        for port := 1235; port <= 1300; port++ {
            altPort := fmt.Sprintf( format: ":%d", port)
            listener, err = net.Listen( network: "tcp", altPort)
            if err == nil {
                err := listener.Close()
                if err != nil {
                    return ""
                }
                return altPort
            }
        }
        log.Fatal( v...: "No available port found in the range 1235-1300")
    }
    defer listener.Close()
    return preferredPort
```

- Hàm getAvailablePort sử dụng để kiểm tra tính khả dụng của một cổng mạng TCP và chọn một cổng thay thế nếu cổng mặc định đã được sử dụng. Đây là một cơ chế tự động để đảm bảo rằng server luôn có thể chạy mà không bị xung đột cổng.

- **Hàm Server**

```go
func server() {  1 usage  new *
    port := getAvailablePort( preferredPort: ":1234")
    fmt.Printf( format: "[S] Using port %s\n", port)

    kv := new(KV)
    kv.data = make(map[string]string)
    newServer := rpc.NewServer()
    errRegister := newServer.Register(kv)
    if errRegister != nil {
        log.Fatal( v...: "[S] Error registering RPC server:", errRegister)
    }

    listener, err := net.Listen( network: "tcp", port)
    if err != nil {
        log.Fatal( v...: "[S] Listen error:", err)
    }
    defer listener.Close()

    fmt.Println( a...: "[S] Wait for connections...")
```

- Hàm server chịu trách nhiệm khởi chạy và vận hành server trong hệ thống Key-Value Store. Đây là thành phần cốt lõi giúp lắng nghe các kết nối từ client, xử lý các yêu cầu thông qua giao thức RPC, và dừng hoạt động khi người dùng yêu cầu.
- Hàm getAvailablePort kiểm tra cổng mặc định :1234 có khả dụng không. Nếu có, trả về cổng này. Nếu không, chọn cổng khả dụng tiếp theo trong khoảng 1235-1300.
- Tạo một đối tượng KV làm kho lưu trữ dữ liệu chính. Sử dụng rpc.NewServer để tạo server RPC.
- Đăng ký các phương thức RPC của KV (như Put, Get) vào server.  Mở một listener TCP để lắng nghe các kết nối từ client trên cổng port. Sử dụng defer listener.Close() để đảm bảo listener được đóng khi server dừng.

```go
    var count int
    var wg sync.WaitGroup
    stop := make(chan struct{}) // Kênh để gửi tín hiệu dừng

    go func() {
        for {
            select {
            case <-stop:
                // Dừng lắng nghe khi nhận tín hiệu
                fmt.Println( a...: "[S] Stopping server...")
                return
            default:
```

- Tạo biến count để đếm số lượng kết nối được xử lý.
- Sử dụng sync.WaitGroup để đồng bộ hóa các goroutines xử lý kết nối.
- Kênh stop để dừng server khi cần.

```go
conn, err := listener.Accept()
if err != nil {
    log.Println( v...: "[S] Connection error:", err)
    continue
}

count++
fmt.Printf( format: "[S] Handle connection %d.\n", count)
wg.Add( delta: 1)
```

- listener.Accept(): Chấp nhận một kết nối TCP từ client và trả về một đối tượng net.Conn đại diện cho kết nối.
- Nếu xảy ra lỗi trong quá trình chấp nhận kết nối, server ghi log lỗi và tiếp tục lắng nghe các kết nối khác.

```go
    count++
    fmt.Printf( format: "[S] Handle connection %d.\n", count)
    wg.Add( delta: 1)

    go func(c net.Conn) {
        defer wg.Done()
        newServer.ServeConn(c)
        c.Close()
    }(conn)
```

- count++ để tăng số lượng kết nối đã được server xử lý.
- Ghi log số thứ tự kết nối đang được xử lý.
- wg.Add(1) để thêm vào sync.WaitGroup để theo dõi tiến trình của các goroutines xử lý kết nối. Đảm bảo server không thoát trước khi tất cả các kết nối hiện tại được xử lý xong.
- Mỗi kết nối từ client được xử lý trong một goroutine riêng biệt để đảm bảo server có thể xử lý nhiều kết nối đồng thời. Tạo routine bằng go func
- defer wg.Done(): khi goroutine hoàn thành công việc, WaitGroup giảm số lượng công việc đang chờ. Đảm bảo không xảy ra deadlock hoặc server dừng trước khi tất cả các kết nối được xử lý.
- newServer.ServeConn(c): xử lý yêu cầu RPC từ client qua kết nối c. Server gọi các phương thức RPC đã đăng ký (ví dụ: Put, Get) để xử lý yêu cầu.
- Cuối cùng, đóng kết nối sau khi xử lý xong để giải phóng tài nguyên.

- **Hàm main**

- Gọi hàm Server() để thực thi

## 3. File Client.go

### a. Chức năng chính

- Gửi yêu cầu Put và Get đến server qua RPC.
- Tạo kết nối mới cho mỗi lần gọi RPC.

### b. Cấu trúc của file

- **Hàm connect**

```
func connect() *rpc.Client {  2 usages  new *
    client, err := rpc.Dial( network: "tcp",  address: "localhost:1234")
    if err != nil {
        log.Fatal( v...: "[C] Dialing error:", err)
    }
    return client
}
```

- Hàm connect được sử dụng để tạo kết nối giữa client và server trong hệ thống sử dụng RPC (Remote Procedure Call). Hàm này đảm bảo rằng client có thể giao tiếp với server thông qua một kênh TCP đã được thiết lập.
- rpc.Dial: Gọi đến server RPC thông qua giao thức TCP với tham số: "tcp": Giao thức mạng được sử dụng (TCP), "localhost:1234": địa chỉ của server (chạy trên localhost và lắng nghe trên cổng 1234). Hàm trả về một đối tượng *rpc.Client nếu kết nối thành công. Nếu không thành công, trả về lỗi (err).

- **Hàm put**

```
func put(key, value string) {  1 usage  new *
    client := connect()
    args := kvs.PutArgs{Key: key, Value: value}
    reply := kvs.PutReply{}
    err := client.Call( serviceMethod: "KV.Put", &args, &reply)
    if err != nil {
        log.Fatal( v...: "[C] Error: ", err)
    }
    errClose := client.Close()

    if errClose != nil {
        log.Println( v...: "[C] Error closing connection:", err)
    }
}
```

- Hàm put được sử dụng bởi client để gửi một yêu cầu RPC tới server nhằm lưu trữ một cặp key-value.
- args: Cấu trúc dữ liệu chứa thông tin của yêu cầu bao gồm cặp key-value.
- reply: Cấu trúc dữ liệu để nhận phản hồi từ server chứa thông tin về trạng thái thực hiện yêu cầu (ví dụ: thành công hoặc thất bại).
- client.Call: Gửi yêu cầu RPC tới server. Phương thức RPC được gọi là "KV.Put" (hàm Put được đăng ký trên server).
- Sau khi hoàn thành yêu cầu RPC, kết nối với server được đóng. Nếu có lỗi xảy ra khi đóng kết nối, log thông báo lỗi.

- **Hàm get**

```go
func get(key string) string {  1 usage  new *
    client := connect()
    args := kvs.GetArgs{Key: key}
    reply := kvs.GetReply{}
    errCall := client.Call( serviceMethod: "KV.Get", &args, &reply)
    if errCall != nil {
        log.Fatal( v...: "[C] Error calling service: ", errCall)
    }
    errClose := client.Close()
    if errClose != nil {
        log.Println( v...: "[C] Error closing connection:", errClose)
    }
    return reply.Value
}
```

- Tương tự như hàm put, hàm get cũng kết nối tới Server và gửi yêu cầu RPC Get tới server.
- Phương thức RPC được gọi là "KV.Get" (hàm Get được định nghĩa trên server).
- Hàm sẽ trả về giá trị tương ứng với key được lưu trên Server

- **Hàm main**

```go
func main() {  new *
    key := "24C11058-NGUYEN_THIEN_PHUC"
    var wg sync.WaitGroup
```

- Biến key là khóa mà các giá trị sẽ được lưu trữ trên server.
- sync.WaitGroup: được sử dụng để chờ tất cả các goroutines hoàn thành trước khi chương trình tiếp tục. Đảm bảo rằng tất cả các lệnh put được thực thi trước khi thực hiện lệnh get.

```go
for i := 0; i < 10; i++ {
    wg.Add( delta: 1)
    go func(n int) {
        defer wg.Done()
        putValue := strconv.Itoa(n)
        fmt.Printf( format: "[C] Calling put(\"%s\",\"%s\")...\n",
            key, putValue)
        put(key, putValue)
    }(i)
}

wg.Wait()
```

- Vòng lặp for thực thi nhiều go routine, vòng lặp 10 lần, mỗi lần thực hiện một goroutine để gọi hàm put với giá trị putValue.

- Trong mỗi vòng lặp:
    - wg.Add(1): Tăng số lượng công việc cần theo dõi trong sync.WaitGroup.

- go func(n int): Tạo một goroutine để thực hiện hàm put song song và truyền giá trị n (giá trị của vòng lặp) vào goroutine.
- defer wg.Done(): Đảm bảo rằng sau khi goroutine hoàn thành, nó sẽ thông báo hoàn thành công việc trong sync.WaitGroup.
- Cuối cùng, gửi yêu cầu RPC Put tới server với khóa key và giá trị putValue.
- wg.Wait(): Chờ tất cả các goroutines thực hiện lệnh put hoàn thành trước khi tiếp tục.

```
value := get(key)
fmt.Printf( format: "[C] get (%s): %s\n", key, value)
```

- get(key): Gửi yêu cầu RPC Get tới server để truy vấn giá trị hiện tại của khóa key.
- Server sẽ trả về giá trị cuối cùng được lưu trữ cho khóa.

# IV. Hướng dẫn sử dụng

## 1. Cài đặt phần mềm

- Phiên bản Go: go1.23.4 darwin/arm64
- Cài đặt code editor như Vscode, Golang, …

## 2. Hướng dẫn sử dụng

**Bước 1:**

- Truy cập vào thư mục /Lab01/Server/Server.go và nhập lệnh "go run Server.go" để start Server. Sau khi start Server, màn hình sẽ in ra như sau:

```
(base) phucnguyen@mac Server % go run .
[S] Using port :1234
[S] Wait for connections...
Press Enter to stop the server...
```

Đây là thông tin port của Server, đồng thời Server bắt đầu lắng nghe kết nối từ Client thông qua port :1234, trong trường hợp người dùng muốn dừng Server thì bấm nút Enter để dừng.

**Bước 2**

- Truy cập vào thư mục /Lab01/Client/Client.go và nhập lệnh "go run Server.go" để start Client.

```
(base) phucnguyen@mac Client % go run .
[C] Calling put("24C11058-NGUYEN_THIEN_PHUC","9")...
[C] Calling put("24C11058-NGUYEN_THIEN_PHUC","3")...
[C] Calling put("24C11058-NGUYEN_THIEN_PHUC","1")...
[C] Calling put("24C11058-NGUYEN_THIEN_PHUC","8")...
[C] Calling put("24C11058-NGUYEN_THIEN_PHUC","0")...
[C] Calling put("24C11058-NGUYEN_THIEN_PHUC","2")...
[C] Calling put("24C11058-NGUYEN_THIEN_PHUC","6")...
[C] Calling put("24C11058-NGUYEN_THIEN_PHUC","7")...
[C] Calling put("24C11058-NGUYEN_THIEN_PHUC","4")...
[C] Calling put("24C11058-NGUYEN_THIEN_PHUC","5")...
[C] get (24C11058-NGUYEN_THIEN_PHUC): 2
```

Đây là kết quả trả về sau khi start Client, trong cơ chế của một key-value store, mỗi khóa chỉ có một giá trị tại một thời điểm. Hàm get truy vấn giá trị hiện tại của khóa 24C11058_NGUYEN_THIEN_PHUC, và 2 là giá trị cuối cùng được ghi bởi một trong các goroutines. Thứ tự hoàn thành của các goroutines là ngẫu nhiên do đó mặc dù 5 là giá trị được put cuối cùng nhưng không phải là giá trị cuối cùng được ghi vào key.
Trong trường hợp người dùng muốn một key lưu được nhiều value thì cần đổi lại cấu trúc dữ liệu từ map[string]string sang map[string][]string – khi này mỗi khóa sẽ lưu trữ một danh sách các giá trị.

## V. Tài liệu tham khảo


[1] Slide bài giảng môn Các hệ thống phân tán - HCMUS
[2] Go Tutorial
[3] net/rpc package