**Amrita Vishwa Vidyapeetham**

**Amrita School of Engineering, Coimbatore**

**Department of Computer Science and Engineering**

**Topic: RPC Implementation in GO**

**-----------------------------------------------------------------------------------------------**

**Sub Code: 18CS632 Sub Title: Distributed Systems**

**Roll No: CB.EN.P2CSE20017 Name: Kannepally Koushik Kumar**

**Assignment: 2 Date: 26/10/2020**

1. Demonstrate client sending remote procedure request and server returning the result to the client stub.

2. Implement one of the variations and demonstrate the same. Demonstrate the differences between the chosen variation and basic RPC.

1. File Structure :

Assignment\_3 rpc

Transcient-Synchronious Communication(Variation 1)

Client

sync\_client.go

Server.go

Transcient-Asynchrounous Communication (Variation 2)

Client

async\_client.go

Server.go

README.docx

-------------------------------------------------------------------------------------------------------------------

**Server :-**

It is a simple server which handles Http request and serves on port No- 4040

Packages used – net/http , net/rpc , crypto/rand ,io

Procedures Defined – GetOTP() -> Gets empty string and pointer to reply message as Parameters. Returns out the otp generated randomly

– ValidateOTP() -> Gets the otp for validation along with its status(expired or active). Returns Validation result to the Client

Built-In Methods Used from RPC Package-

1. **Register()** -> Accepts an Interface as Parameters - > Stores all Suitable methods into the interface depending on type of interface we pass

Example:- Here we pass int Api interface into Regeister Function

type API int

    api := new(API)

    err := rpc.Register(api)

It adds up another interface which stores the sent interface type,value,name, suitable methods to the client which they can call remotely to access procedures that are written inside the server

Ex:- Call() method

s := new(service)

    s.typ = reflect.TypeOf(rcvr)

    s.rcvr = reflect.ValueOf(rcvr)

    sname := reflect.Indirect(s.rcvr).Type().Name()

s.method = suitableMethods(s.typ, true)

s.method = suitableMethods(s.typ, true)

Here is the function how suitable methods are given :-

func suitableMethods(typ reflect.Type, reportErr bool) map[string]\*methodType {

    methods := make(map[string]\*methodType)

    for m := 0; m < typ.NumMethod(); m++ {

        method := typ.Method(m)

        mtype := method.Type

        mname := method.Name

        // Method must be exported.

        if method.PkgPath != "" {

            continue

        }

        // Method needs three ins: receiver, \*args, \*reply.

        if mtype.NumIn() != 3 {

            if reportErr {

                log.Printf("rpc.Register: method %q has %d input parameters; needs exactly three\n", mname, mtype.NumIn())

            }

            continue

        }

        // First arg need not be a pointer.

        argType := mtype.In(1)

        if !isExportedOrBuiltinType(argType) {

            if reportErr {

                log.Printf("rpc.Register: argument type of method %q is not exported: %q\n", mname, argType)

            }

            continue

        }

        // Second arg must be a pointer.

        replyType := mtype.In(2)

        if replyType.Kind() != reflect.Ptr {

            if reportErr {

                log.Printf("rpc.Register: reply type of method %q is not a pointer: %q\n", mname, replyType)

            }

            continue

        }

        // Reply type must be exported.

        if !isExportedOrBuiltinType(replyType) {

            if reportErr {

                log.Printf("rpc.Register: reply type of method %q is not exported: %q\n", mname, replyType)

            }

            continue

        }

        // Method needs one out.

        if mtype.NumOut() != 1 {

            if reportErr {

                log.Printf("rpc.Register: method %q has %d output parameters; needs exactly one\n", mname, mtype.NumOut())

            }

            continue

        }

        // The return type of the method must be error.

        if returnType := mtype.Out(0); returnType != typeOfError {

            if reportErr {

                log.Printf("rpc.Register: return type of method %q is %q, must be error\n", mname, returnType)

            }

            continue

        }

        methods[mname] = &methodType{method: method, ArgType: argType, ReplyType: replyType}

    }

    return methods

}

1. **HandleHTTP()** -> It says to handle Http request on Default path

func HandleHTTP() {

    DefaultServer.HandleHTTP(DefaultRPCPath, DefaultDebugPath)

}

Where, The defaults paths are

const (

    // Defaults used by HandleHTTP

    DefaultRPCPath   = "/\_goRPC\_"

    DefaultDebugPath = "/debug/rpc"

)

**Clients :-**

Clients make rpc calls to the procedures running on server by sending http requests and access them remotely.

Pakages used – net/rpc , time, fmt

Built-In Methods Used from RPC Package-

1. **DailHTTP(“network” , IP\_of\_Server: port\_NO)** - >

This function in-turn calls – Dail() function from “net” package-> which returns out connection object

This connection object has all the suitable methods required to invoke the procedures that are written at server side which is running at some port number as a process

func DialHTTPPath(network, address, path string) (\*Client, error) {

    var err error

    conn, err := net.Dial(network, address)

    if err != nil {

        return nil, err

    }

    io.WriteString(conn, "CONNECT "+path+" HTTP/1.0\n\n")

}

Now we can send HTTP requests to the server to which we are connected by this connection object

client, err := rpc.DialHTTP("tcp", "192.168.0.114:4040")

Since we are storing that connection object inside the client variable, We are now able to call the procedures that are written as server side using Call() method and store responses.

Ex:-

client.Call("API.GetOTP", "", &reply)

client.Call("API.ValidateOTP", rotp, &response)

Instructions to Execute:-

* Go to the directory where server.go is present in Terminal and type the command **go run server.go**
* Open the Client program in any editor.
* At line number 18, Change the IP addr with the IP of the system on which you are running the **server.go**

**Ex:-** client, err := rpc.DialHTTP("tcp", "192.168.0.114:4040")

* Now open another the terminal and go to directory where client program is present and type command

Go run {client\_program\_name}.go

Here, client\_program\_name is -> sync\_client.go / async\_client.go

1. **Variations Implemented:-**
2. Transcient- Synchronous communication:-

-> Here the client waits for the response(Which is otp) that needs to come from the server . And then starts the timer . We need to send the otp to the server within the time limit given or it expires.

Which shows synchrounous behaviour i.e., the client is waiting untill the response comes and blocking all other code at client side

-> Here we don’t have any message queuing system so it cant be persistent

* Here is the timer function which runs in background

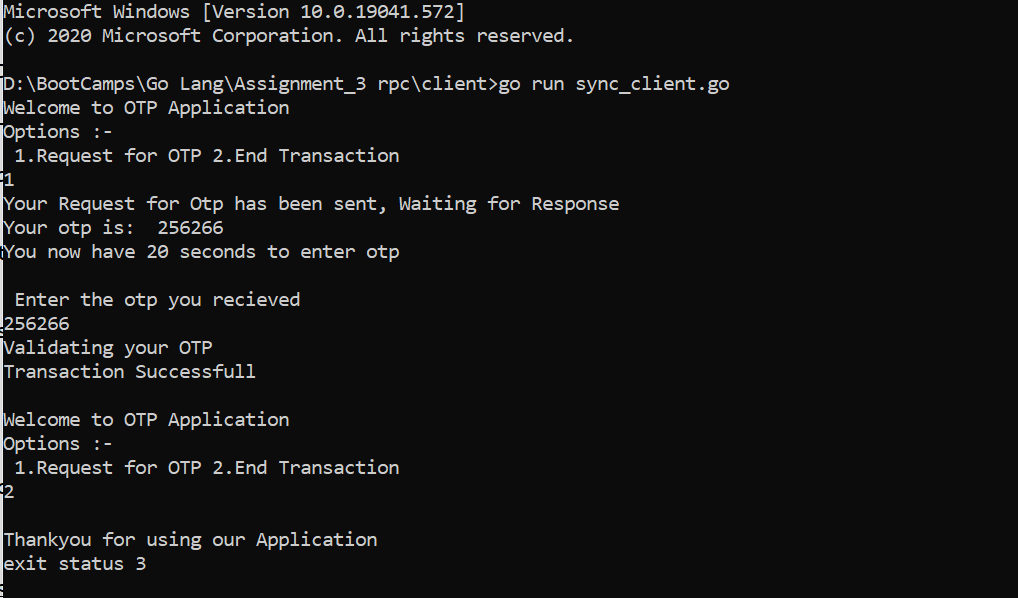
afterFuncTimer := time.AfterFunc(time.Second\*20, func() {

                //fmt.Println("Your time is completed")  -> uncomment this to debug when the timer is finishing

                status = "expired"

            })

**Sample output ScreenShots:-**

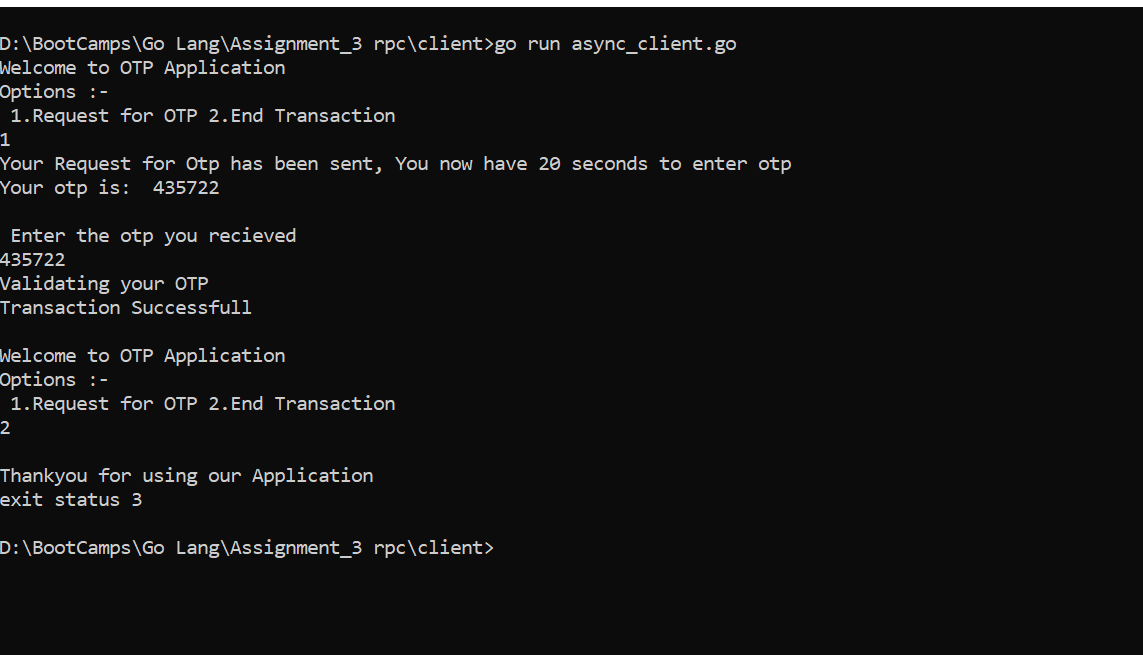


1. Transicent – Asynchronous Communication:-

-> Here the client doesnot wait for the response and starts running the timer as soon as request is sent.

-> Since there is no message queuing system this can be transcient way of communication.

**Sample Ouput Screenshots:-**

****

**Note :-**

If we are using asynchronous type of communication the timer starts before recieveing the response.

If the is any delay in the network and we have less time to enter otp , the timer expires before the otp arrives to the client. For which we need to initiate the transcation again.

This can be a disadvantage in banking applications since the functions happen asynchronously there.