

# RPCs in Go

Feb 9<sup>th</sup> & 10<sup>th</sup>, 2022

# Outline

RPC Overview

Example: Writing an RPC server in Go

# RPC Overview

# Remote Procedure Call (RPC)

- Execute a procedure on a remote process (e.g on another server) as if it was local
- Request-response interface
  - Request: arguments to remote procedure
  - Response: return values of remote procedure
- Examples: client-server, master-worker, peer-peer communication

# Example: Master-Worker

```
Master {  
    func LaunchTasks() {  
        for worker in workers {  
            // want to call Worker.RunTask(...)  
        }  
    }  
}
```

```
Worker {  
    func RunTask(index) result {  
        // ...  
    }  
}
```

# Example: Master-Worker

```
Master {  
    func LaunchTasks() {  
        for worker in workers {  
            index = worker.Index  
            address = worker.Address  
            request = MakeRequest(index)  
            response = sendRPC("RunTask", address, request)  
            result = response.Result  
            handleResult(result)  
        }  
    }  
}
```

```
Worker {  
    func RunTask(index) result {  
        // ...  
    }  
}
```

What is the problem?  
How is performance?

# Asynchronous RPC

Key Idea: Await RPC response in a separate thread

Multiple ways to implement this:

1. Pass a *callback* to RPC that will be invoked later

# Asynchronous RPC

Await RPC response in a separate thread

Multiple ways to implement this:

1. Pass a *callback* to RPC that will be invoked later

```
func handleResponse {  
    ...  
    // e.g process result and notify the master  
}  
  
sendRPC("RunTask", address, request, handleResponse)
```



# Asynchronous RPC

Await RPC response in a separate thread

Multiple ways to implement this:

1. Pass a *callback* to RPC that will be invoked later
2. Use *channels* to communicate RPC reply back to main thread

# Asynchronous RPC

Await RPC response in a separate thread

Multiple ways to implement this:

1. Pass a *callback* to RPC that will be invoked later
2. Use *channels* to communicate RPC reply back to main thread

```
for _, worker := range workers {
    go func() {
        channel <- sendRPC("RunTask", address, request)
    }()
}
select {
    case res := <-channel:
        handleResponse(res)
    default:
        // do other stuff
}
```

What's an example application where we would want asynchronous RPCs?

# Writing a RPC server in GO

# RPC Implementations in Go

- There are 3 types of RPC implementations in Go's built-in library
  - `net/rpc`
  - `net/rpc/jsonrpc`
  - `gRPC`

# RPCs in GO (net/rpc server)

- Write stub receiver methods in the form:

```
func (t *T) MethodName(args T1, reply *T2) error
```

- Create a server
  - Create a TCP server (or some other types of server to receive data)
  - Create a listener that will handle RPCs
  - Register the listener and accept inbound RPC
- See <https://golang.org/pkg/net/rpc/> for more details

# Go example: Word count server

```
type WordCountServer struct {  
    addr string  
}
```

```
type WordCountRequest struct {  
    Input string  
}
```

```
type WordCountReply struct {  
    Counts map[string]int  
}
```

```
func (*WordCountServer) Compute(  
    request WordCountRequest,  
    reply *WordCountReply) error {  
    counts := make(map[string]int)  
    input := request.Input  
    tokens := strings.Fields(input)  
    for _, t := range tokens {  
        counts[t] += 1  
    }  
    reply.Counts = counts  
    return nil  
}
```

Step 1: write the stub  
function

# Go example: Word count server

```
type WordCountServer struct {  
    addr string  
}
```

```
type WordCountRequest struct {  
    Input string  
}
```

```
type WordCountReply struct {  
    Counts map[string]int  
}
```

```
func (*WordCountServer) Compute(  
    request WordCountRequest,  
    reply *WordCountReply) error {  
    counts := make(map[string]int)  
    input := request.Input  
    tokens := strings.Fields(input)  
    for _, t := range tokens {  
        counts[t] += 1  
    }  
    reply.Counts = counts  
    return nil  
}
```

Step 1: write the stub  
function



# Go example: Word count server

```
func (server *WordCountServer) Listen() {  
    rpc.Register(server)  
    listener, err := net.Listen("tcp", server.addr)  
    checkError(err)  
    go func() {  
        rpc.Accept(listener)  
    }()  
}
```

Step 2.1: create a server

# Go example: Word count server

```
func (server *WordCountServer) Listen() {  
    rpc.Register(server)  
    listener, err := net.Listen("tcp", server.addr)  
    checkError(err)  
    go func() {  
        rpc.Accept(listener)  
    }()  
}
```

Step 2.2: create a listener that handles RPCs

# Go example: Word count server

```
func (server *WordCountServer) Listen() {  
    rpc.Register(server)  
    listener, err := net.Listen("tcp", server.addr)  
    checkError(err)  
    go func() {  
        rpc.Accept(listener)  
    }()  
}
```

Step 2.3: register the listener  
and accept inbound RPCs

## RPCs in GO (net/rpc client)

- Create a client
- Issue a RPC call
- Unpack return value

# Go example: Word count client

```
func makeRequest(input string, serverAddr string) (map[string]int, error) {  
    client, err := rpc.Dial("tcp", serverAddr)  
    checkError(err)  
    args := WordCountRequest{input}  
    reply := WordCountReply{make(map[string]int)}  
    err = client.Call("WordCountServer.Compute", args, &reply)  
    if err != nil {  
        return nil, err  
    }  
    return reply.Counts, nil  
}
```

Step 1: create a client

# Go example: Word count client

```
func makeRequest(input string, serverAddr string) (map[string]int, error) {
    client, err := rpc.Dial("tcp", serverAddr)
    checkError(err)
    args := WordCountRequest{input}
    reply := WordCountReply{make(map[string]int)}
    err = client.Call("WordCountServer.Compute", args, &reply)
    if err != nil {
        return nil, err
    }
    return reply.Counts, nil
}
```

Step 2.1: create the RPC arguments

# Go example: Word count client

```
func makeRequest(input string, serverAddr string) (map[string]int, error) {  
    client, err := rpc.Dial("tcp", serverAddr)  
    checkError(err)  
    args := WordCountRequest{input}  
    reply := WordCountReply{make(map[string]int)}  
    err = client.Call("WordCountServer.Compute", args, &reply)  
    if err != nil {  
        return nil, err  
    }  
    return reply.Counts, nil  
}
```

Step 2.2: Make a RPC call

# Go example: Word count client-server

```
func main() {  
    serverAddr := "localhost:8888"  
    server := WordCountServer{serverAddr}  
    server.Listen()  
    input1 := "hello I am good hello bye bye bye good night hello"  
    wordcount, err := makeRequest(input1, serverAddr)  
    checkError(err)  
    fmt.Printf("Result: %v\n", wordcount)  
}
```

Step 3: Unpack return values

```
Result: map[hello:3 I:1 am:1 good:2 bye:4 night:1]
```



# Is this synchronous or asynchronous?

```
func makeRequest(input string, serverAddr string) (map[string]int, error) {  
    client, err := rpc.Dial("tcp", serverAddr)  
    checkError(err)  
    args := WordCountRequest{input}  
    reply := WordCountReply{make(map[string]int)}  
    err = client.Call("WordCountServer.Compute", args, &reply)  
    if err != nil {  
        return nil, err  
    }  
    return reply.Counts, nil  
}
```

# Making client asynchronous

```
func makeRequest(input string, serverAddr string) chan Result {
    client, err := rpc.Dial("tcp", serverAddr)
    checkError(err)
    args := WordCountRequest{input}
    reply := WordCountReply{make(map[string]int)}
    ch := make(chan Result)
    go func() {
        err := client.Call("WordCountServer.Compute", args, &reply)
        if err != nil {
            ch <- Result{nil, err} // something went wrong
        } else {
            ch <- Result{reply.Counts, nil} // success
        }
    }()
    return ch
}
```

# Making client asynchronous

```
func makeRequest(input string, serverAddr string) *Call {
    client, err := rpc.Dial("tcp", serverAddr)
    checkError(err)
    args := WordCountRequest{input}
    reply := WordCountReply{make(map[string]int)}
    return client.Go("WordCountServer.Compute", args, &reply, nil)
}
```

```
call := makeRequest(...)
<-call.Done
checkError(call.Error)
handleReply(call.Reply)
```

# Go's net/rpc is at-most-once

- Opens a TCP connection and writes the request
  - TCP may retransmit but server's TCP receiver will **filter out duplicates internally**, with sequence numbers
  - No retry in Go RPC code (i.e will **not** create a second TCP connection)
- However, Go RPC returns an error if it doesn't get a reply
  - Perhaps after a TCP timeout
  - Perhaps server didn't see the request
  - Perhaps server processed request but server or network failed before reply came back

# RPC and Assignment 1 and 2

- Go's RPC **isn't enough** for Assignments 1 and 2
  - It only applies to a single RPC call
  - If worker doesn't respond, master **re-sends** to another (e.g handling worker failures in part D of assignment 1-3)
    - Go RPC **can't detect** this kind of duplicate
  - **Breaks at-most-once semantics**
    - No problem in Assignments 1 and 2 (handles at application level)
- In Assignment 3, **you** will explicitly detect duplicates using techniques we've talked about in lectures