Ex06 Middleware

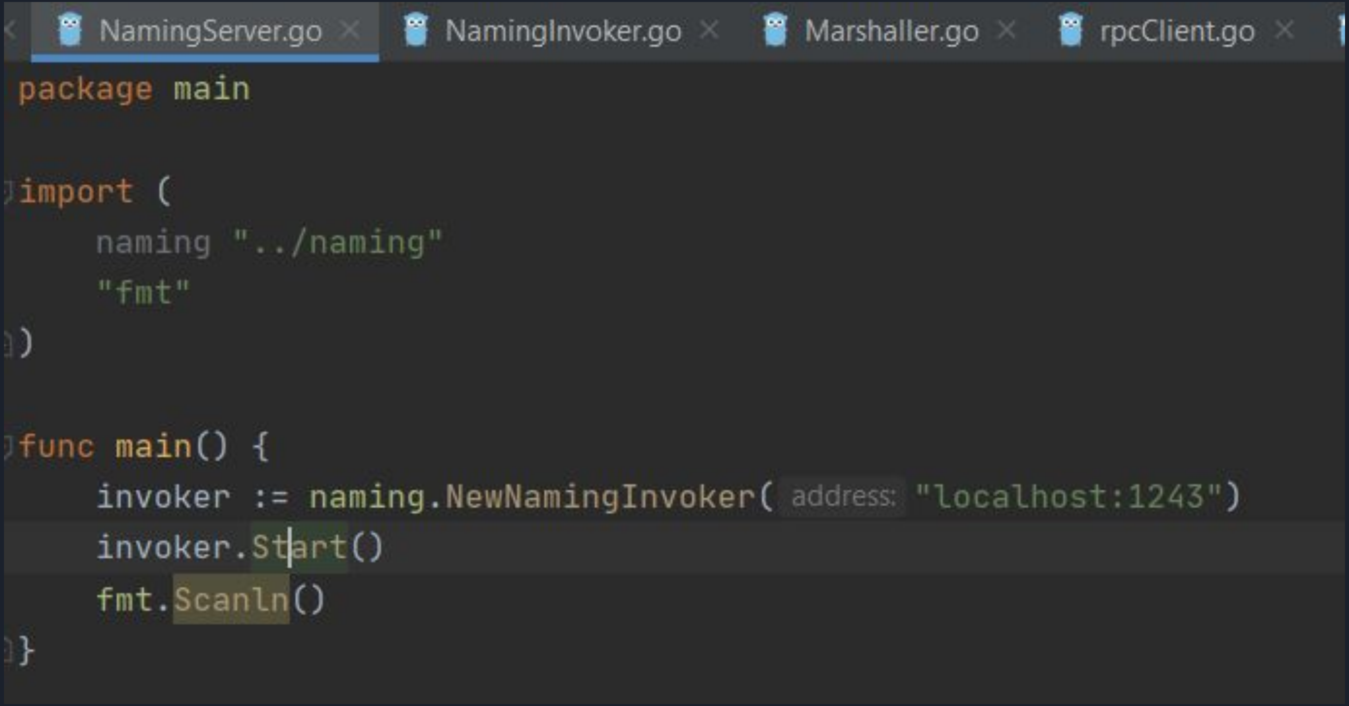
Aplicação

- Vanets
- Carros se registram na RSU mais próxima informando a ‘lane’ que eles estão. Podendo sempre registrar quando mudam de ‘lane’
- Caso um carro avise de algum evento naquela ‘lane’, todos os carros que estão naquela lane recebem o evento, e tomam alguma ação (Freiar, mudar de lane, mudar de rota, etc)

Experimento - Definição

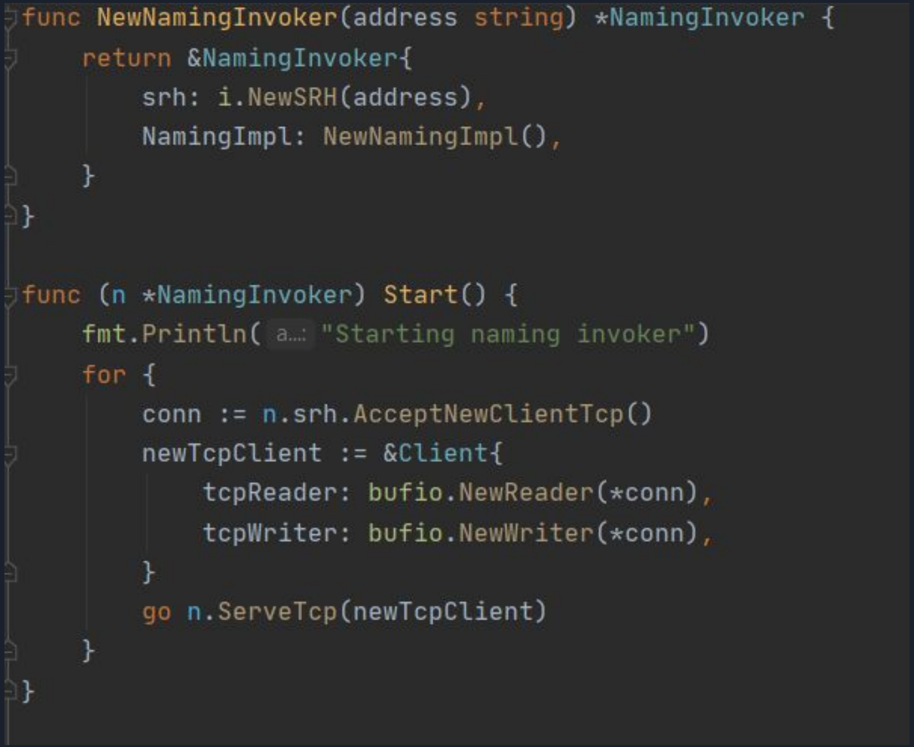
- Não abro e fecho a conexão para cada request
- Envio de uma mensagem

NamingServer



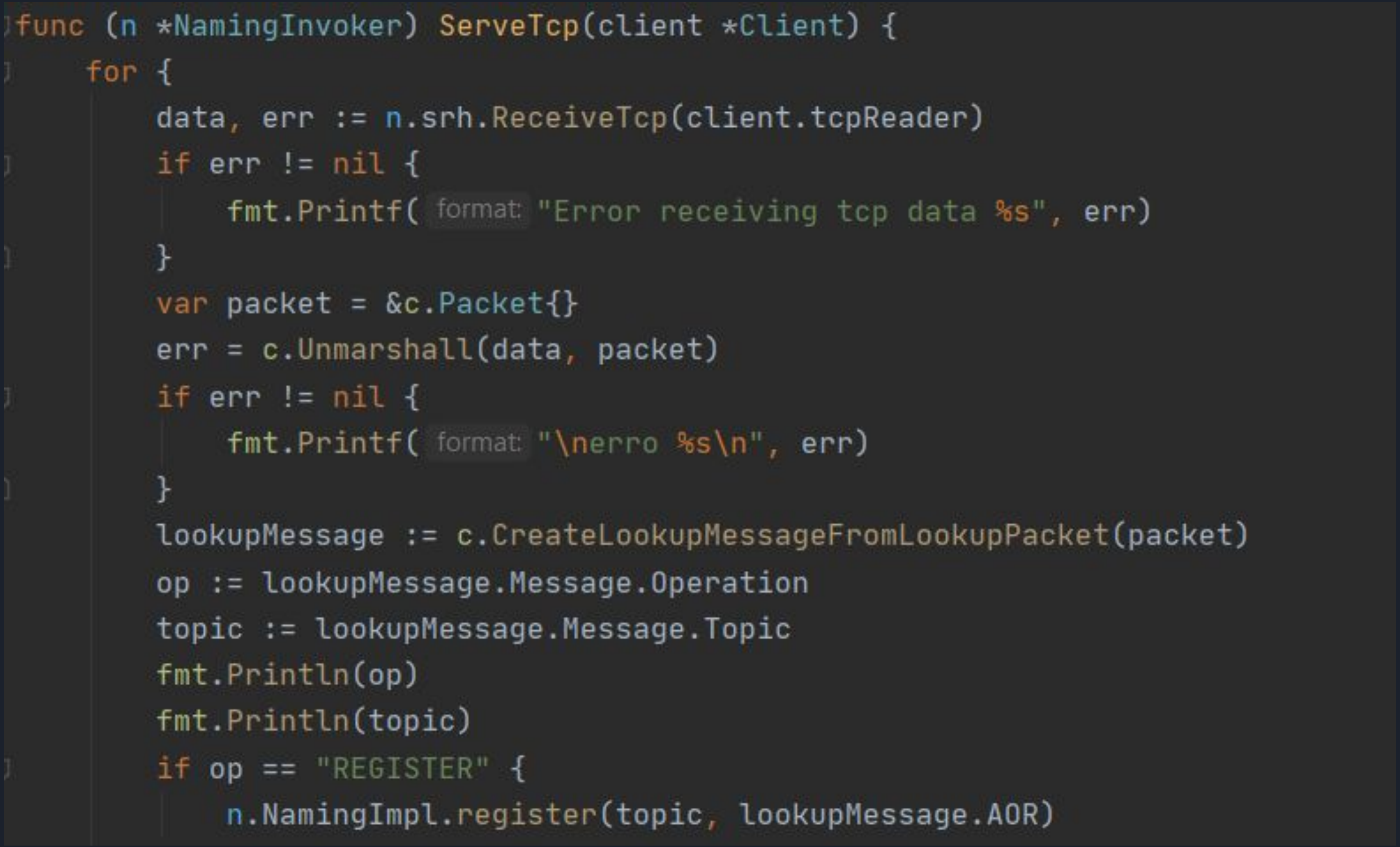
```go
package main

import (
	naming "../naming"
	"fmt"
)

func main() {
	invoker := naming.NewNamingInvoker( address: "localhost:1243")
	invoker.Start()
	fmt.Scanln()
}
```

NamingInvoker



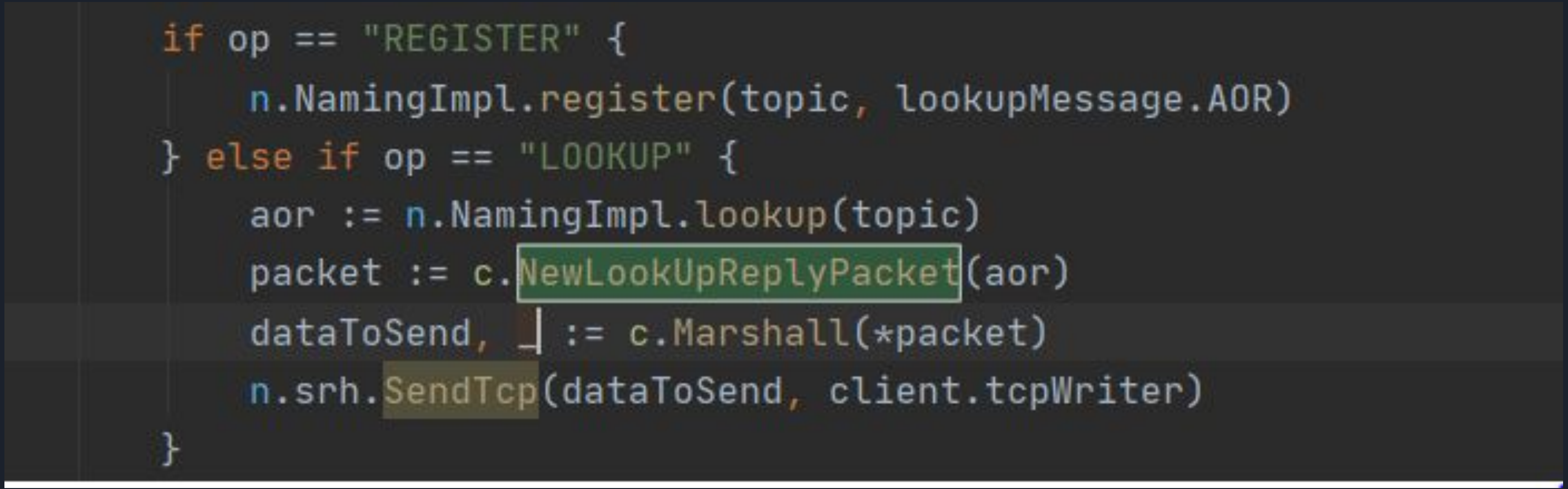
```go
func NewNamingInvoker(address string) *NamingInvoker {
    return &NamingInvoker{
        srh: i.NewSRH(address),
        NamingImpl: NewNamingImpl(),
    }
}

func (n *NamingInvoker) Start() {
    fmt.Println( a...: "Starting naming invoker")
    for {
        conn := n.srh.AcceptNewClientTcp()
        newTcpClient := &Client{
            tcpReader: bufio.NewReader(*conn),
            tcpWriter: bufio.NewWriter(*conn),
        }
        go n.ServeTcp(newTcpClient)
    }
}
```

NamingInvoker



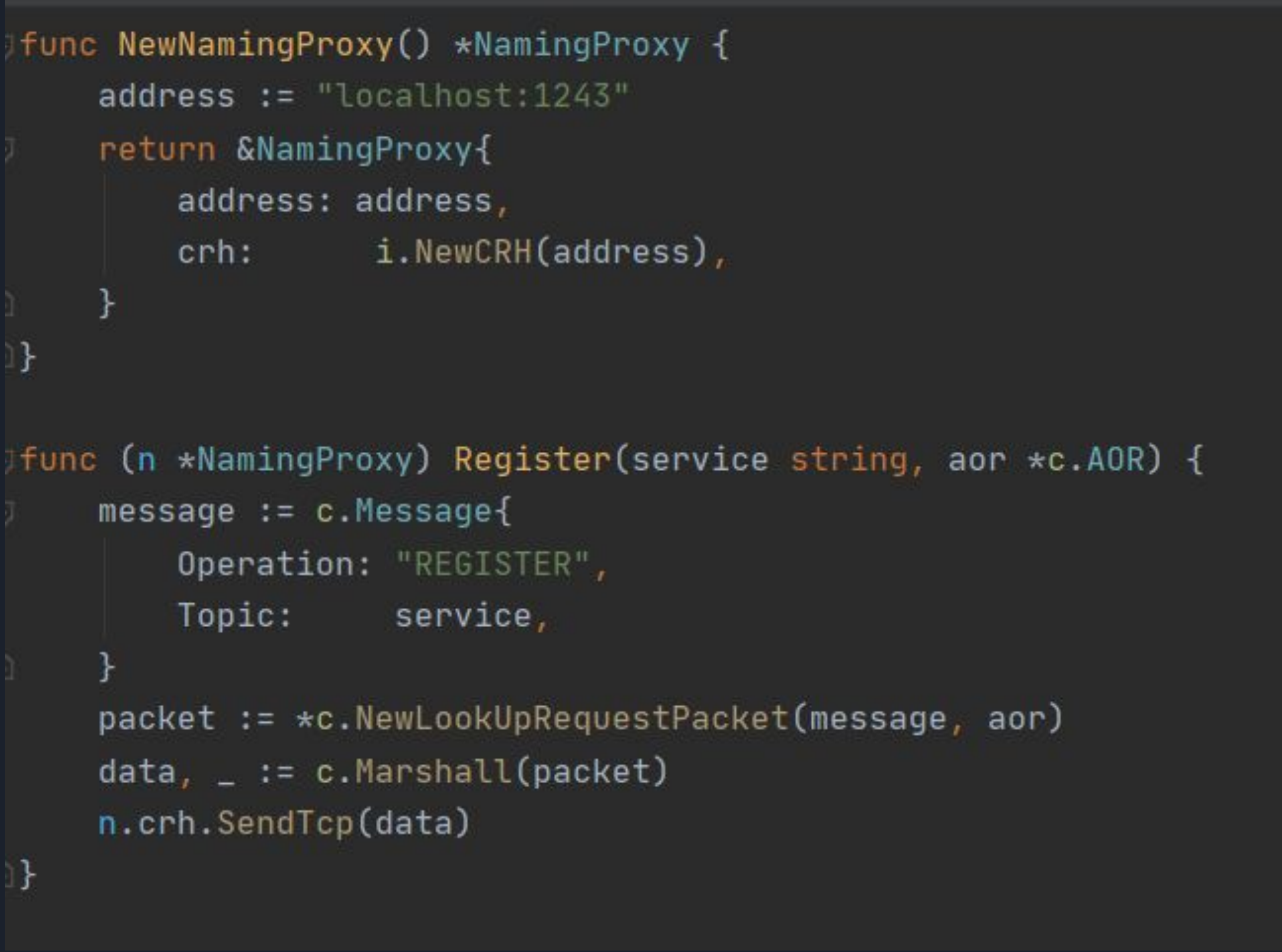
```go
func (n *NamingInvoker) ServeTcp(client *Client) {
    for {
        data, err := n.srh.ReceiveTcp(client.tcpReader)
        if err != nil {
            fmt.Printf( format: "Error receiving tcp data %s", err)
        }
        var packet = &c.Packet{}
        err = c.Unmarshall(data, packet)
        if err != nil {
            fmt.Printf( format: "\nerro %s\n", err)
        }
        lookupMessage := c.CreateLookupMessageFromLookupPacket(packet)
        op := lookupMessage.Message.Operation
        topic := lookupMessage.Message.Topic
        fmt.Println(op)
        fmt.Println(topic)
        if op == "REGISTER" {
            n.NamingImpl.register(topic, lookupMessage.AOR)
```

NamingInvoker



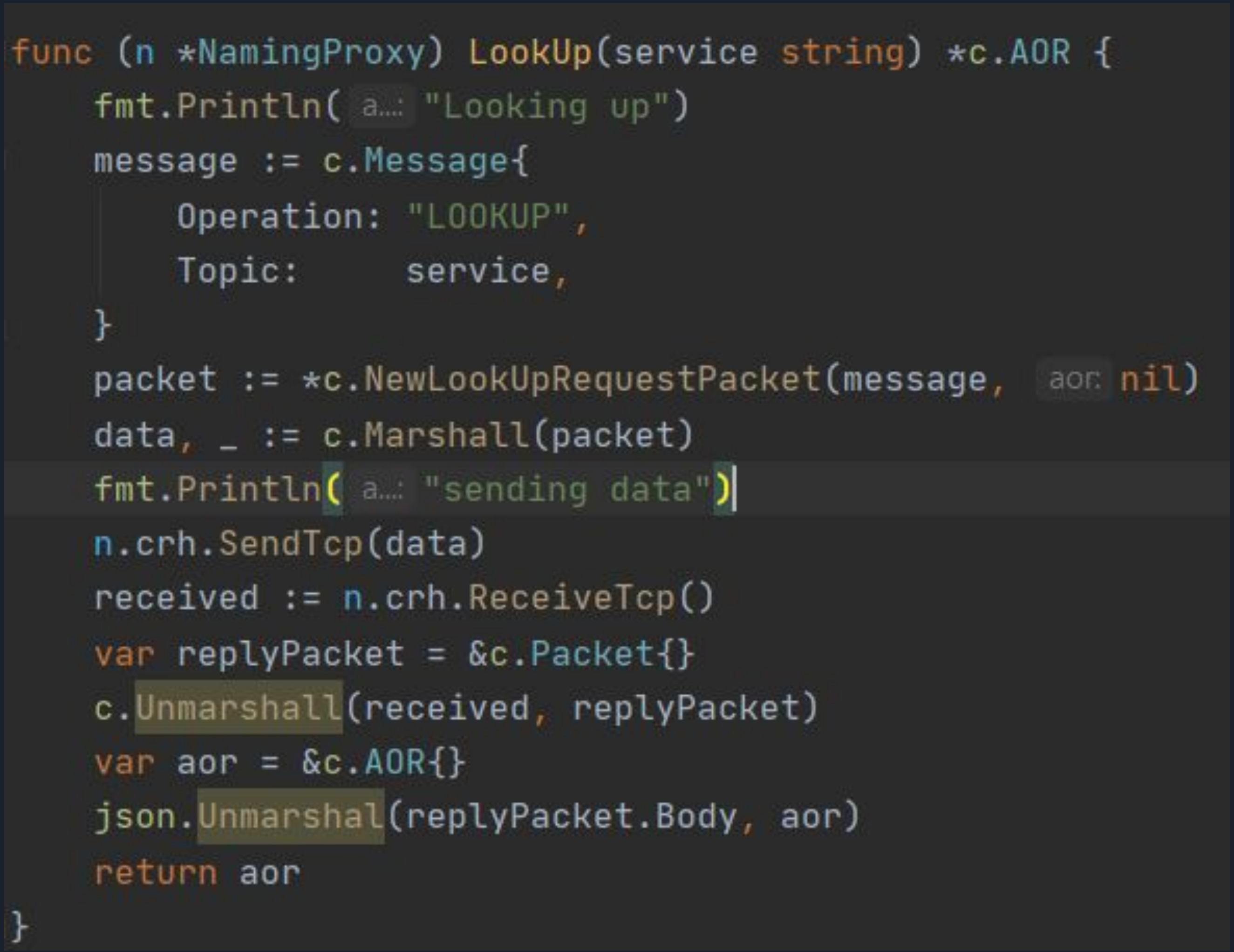
```
if op == "REGISTER" {
    n.NamingImpl.register(topic, lookupMessage.AOR)
} else if op == "LOOKUP" {
    aor := n.NamingImpl.lookup(topic)
    packet := c.NewLookUpReplyPacket(aor)
    dataToSend, _ := c.Marshall(*packet)
    n.srh.SendTcp(dataToSend, client.tcpWriter)
}
```

NamingProxy



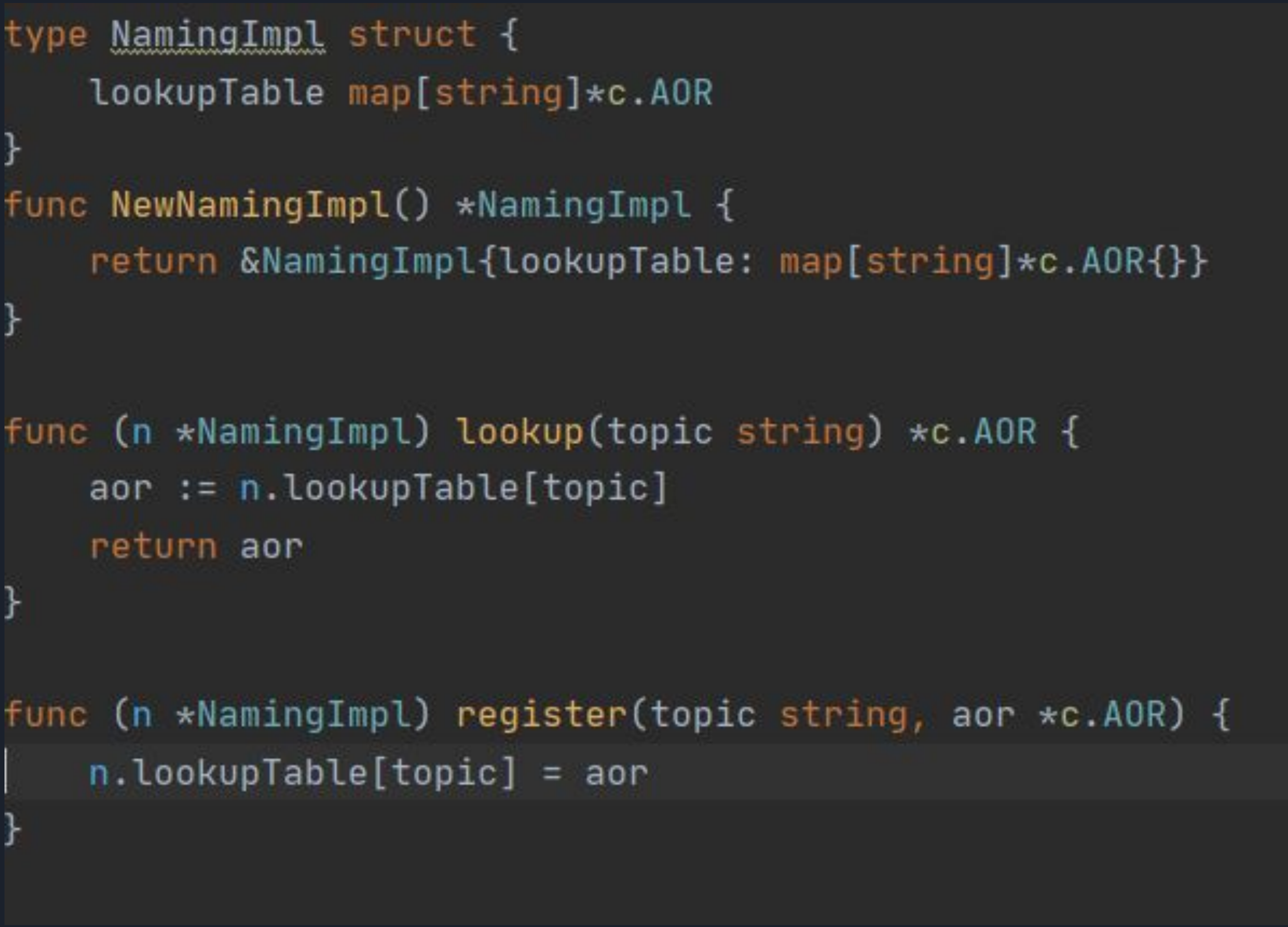
```go
func NewNamingProxy() *NamingProxy {
    address := "localhost:1243"
    return &NamingProxy{
        address: address,
        crh:     i.NewCRH(address),
    }
}

func (n *NamingProxy) Register(service string, aor *c.AOR) {
    message := c.Message{
        Operation: "REGISTER",
        Topic:     service,
    }
    packet := *c.NewLookUpRequestPacket(message, aor)
    data, _ := c.Marshall(packet)
    n.crh.SendTcp(data)
}
```

NamingProxy



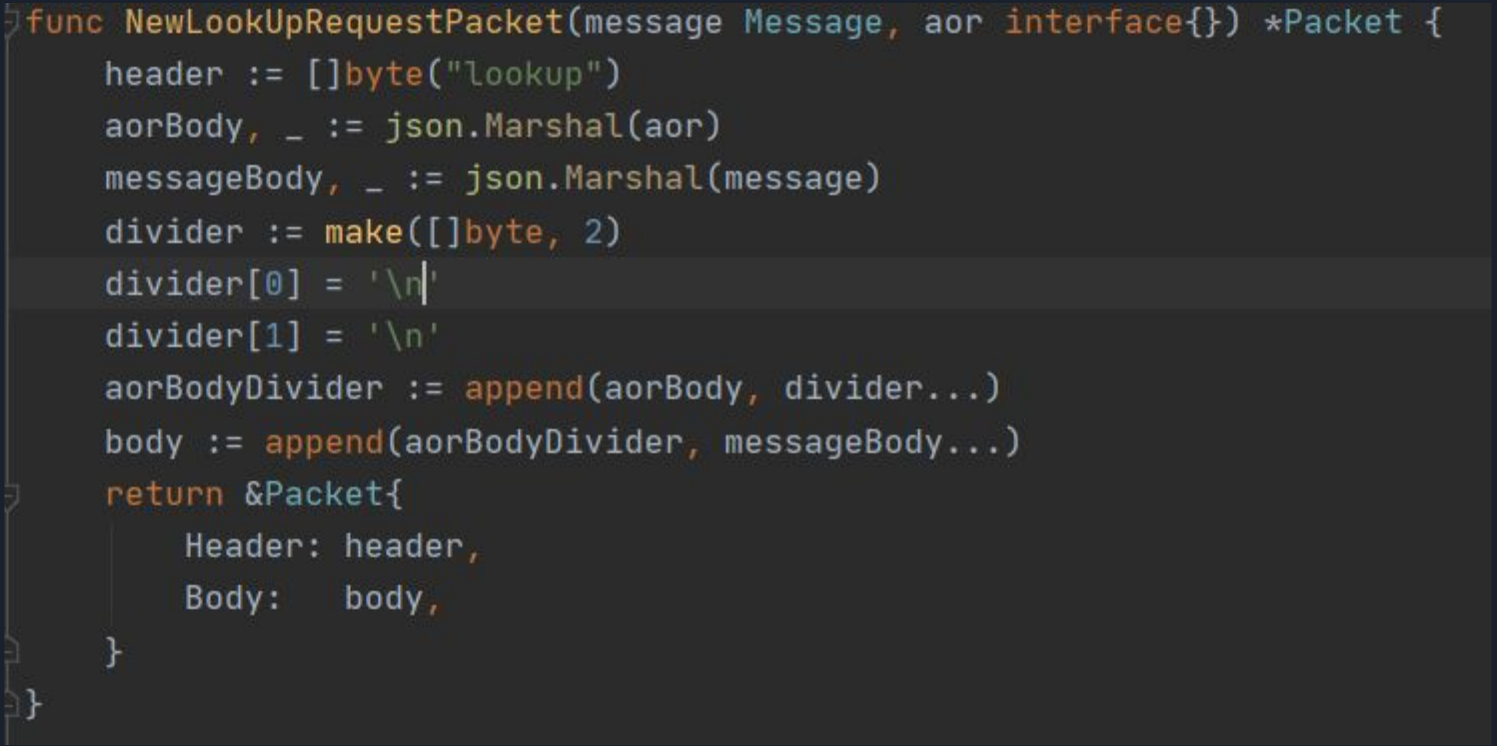
```
func (n *NamingProxy) LookUp(service string) *c.AOR {
    fmt.Println( a...: "Looking up")
    message := c.Message{
        Operation: "LOOKUP",
        Topic:     service,
    }
    packet := *c.NewLookUpRequestPacket(message,  aor: nil)
    data, _ := c.Marshall(packet)
    fmt.Println( a...: "sending data")
    n.crh.SendTcp(data)
    received := n.crh.ReceiveTcp()
    var replyPacket = &c.Packet{}
    c.Unmarshall(received, replyPacket)
    var aor = &c.AOR{}
    json.Unmarshal(replyPacket.Body, aor)
    return aor
}
```

NamingImpl



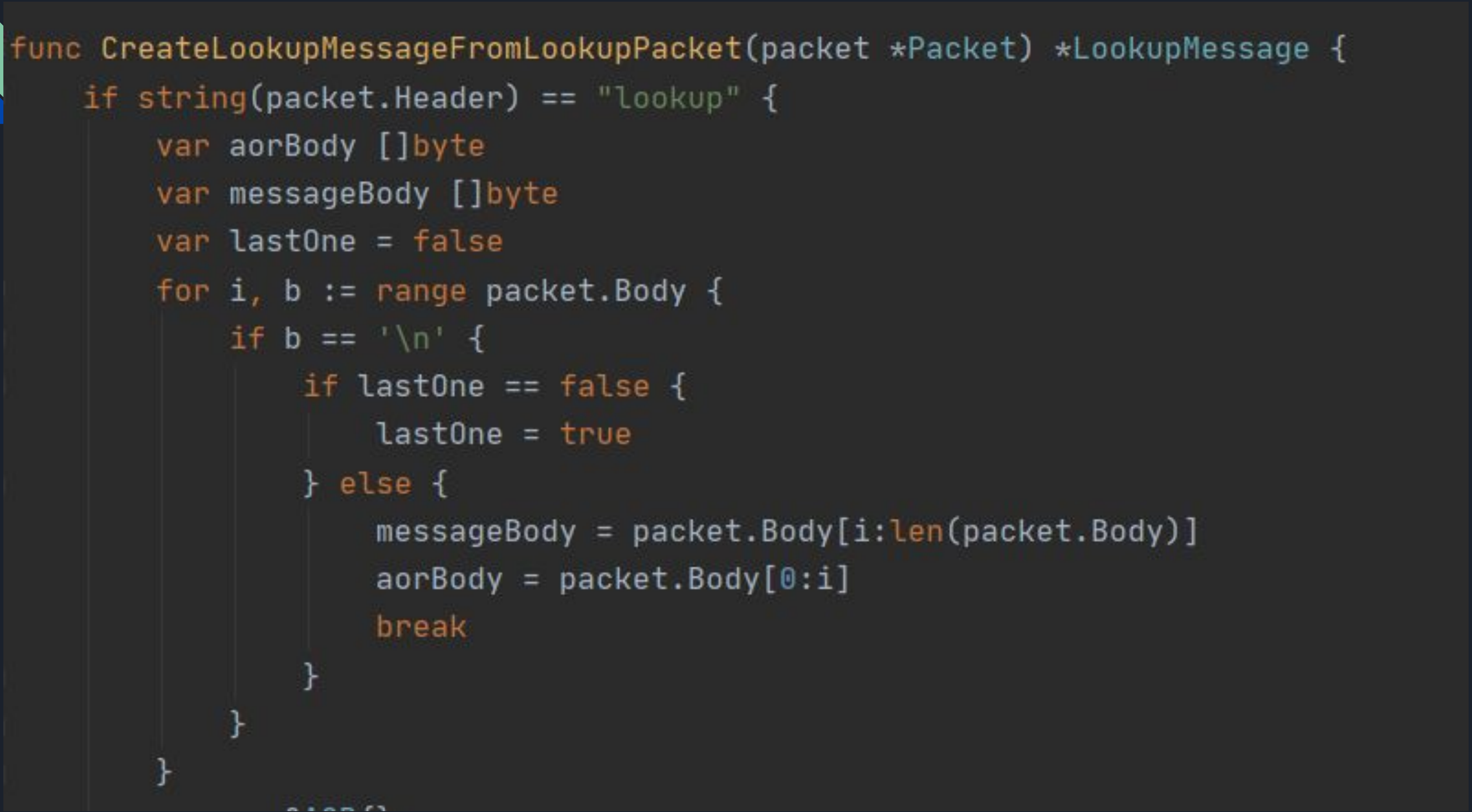
```go
type NamingImpl struct {
    lookupTable map[string]*c.AOR
}
func NewNamingImpl() *NamingImpl {
    return &NamingImpl{lookupTable: map[string]*c.AOR{}}
}

func (n *NamingImpl) lookup(topic string) *c.AOR {
    aor := n.lookupTable[topic]
    return aor
}

func (n *NamingImpl) register(topic string, aor *c.AOR) {
    n.lookupTable[topic] = aor
}
```

Packet



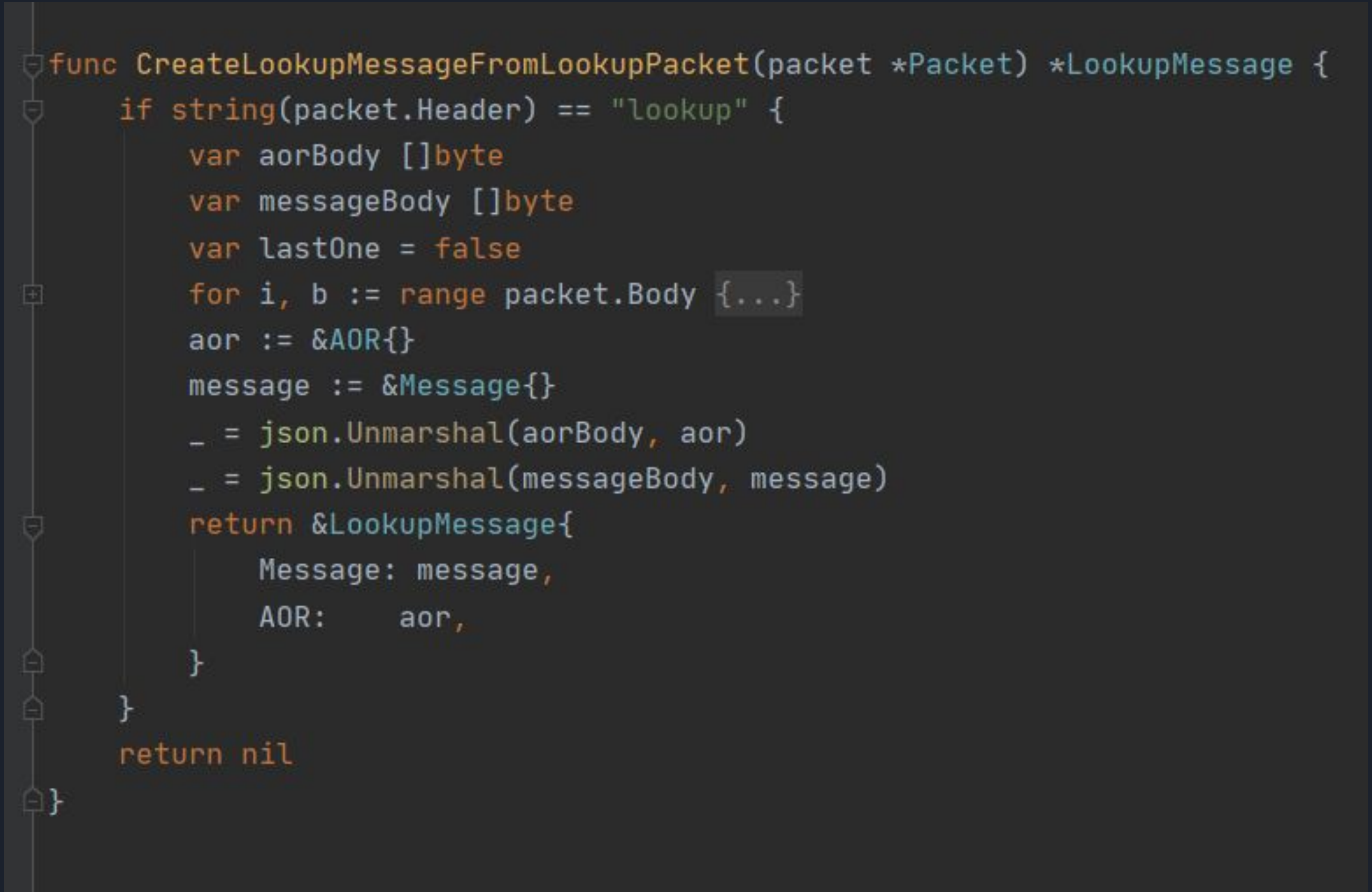
```go
func NewLookUpRequestPacket(message Message, aor interface{}) *Packet {
	header := []byte("lookup")
	aorBody, _ := json.Marshal(aor)
	messageBody, _ := json.Marshal(message)
	divider := make([]byte, 2)
	divider[0] = '\n'
	divider[1] = '\n'
	aorBodyDivider := append(aorBody, divider...)
	body := append(aorBodyDivider, messageBody...)
	return &Packet{
		Header: header,
		Body:   body,
	}
}
```

Packet



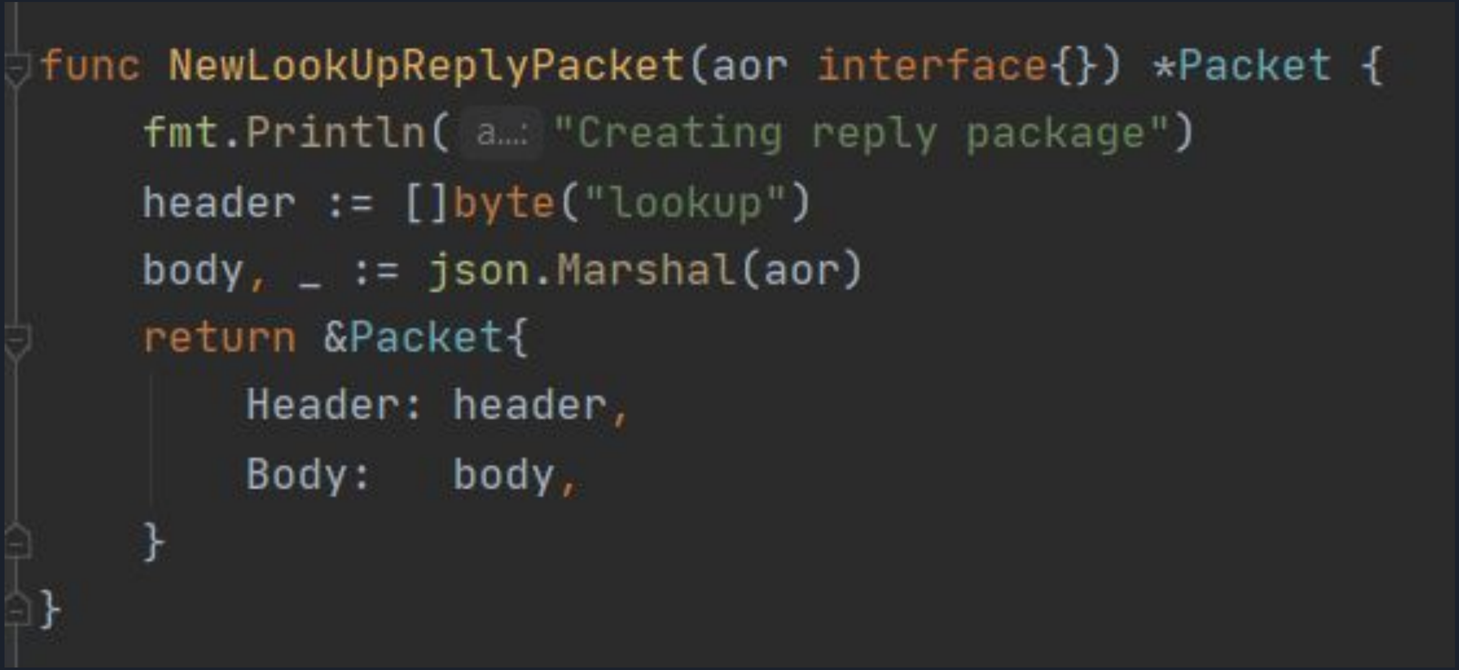
```go
func CreateLookupMessageFromLookupPacket(packet *Packet) *LookupMessage {
    if string(packet.Header) == "lookup" {
        var aorBody []byte
        var messageBody []byte
        var lastOne = false
        for i, b := range packet.Body {
            if b == '\n' {
                if lastOne == false {
                    lastOne = true
                } else {
                    messageBody = packet.Body[i:len(packet.Body)]
                    aorBody = packet.Body[0:i]
                    break
                }
            }
        }
```

Packet



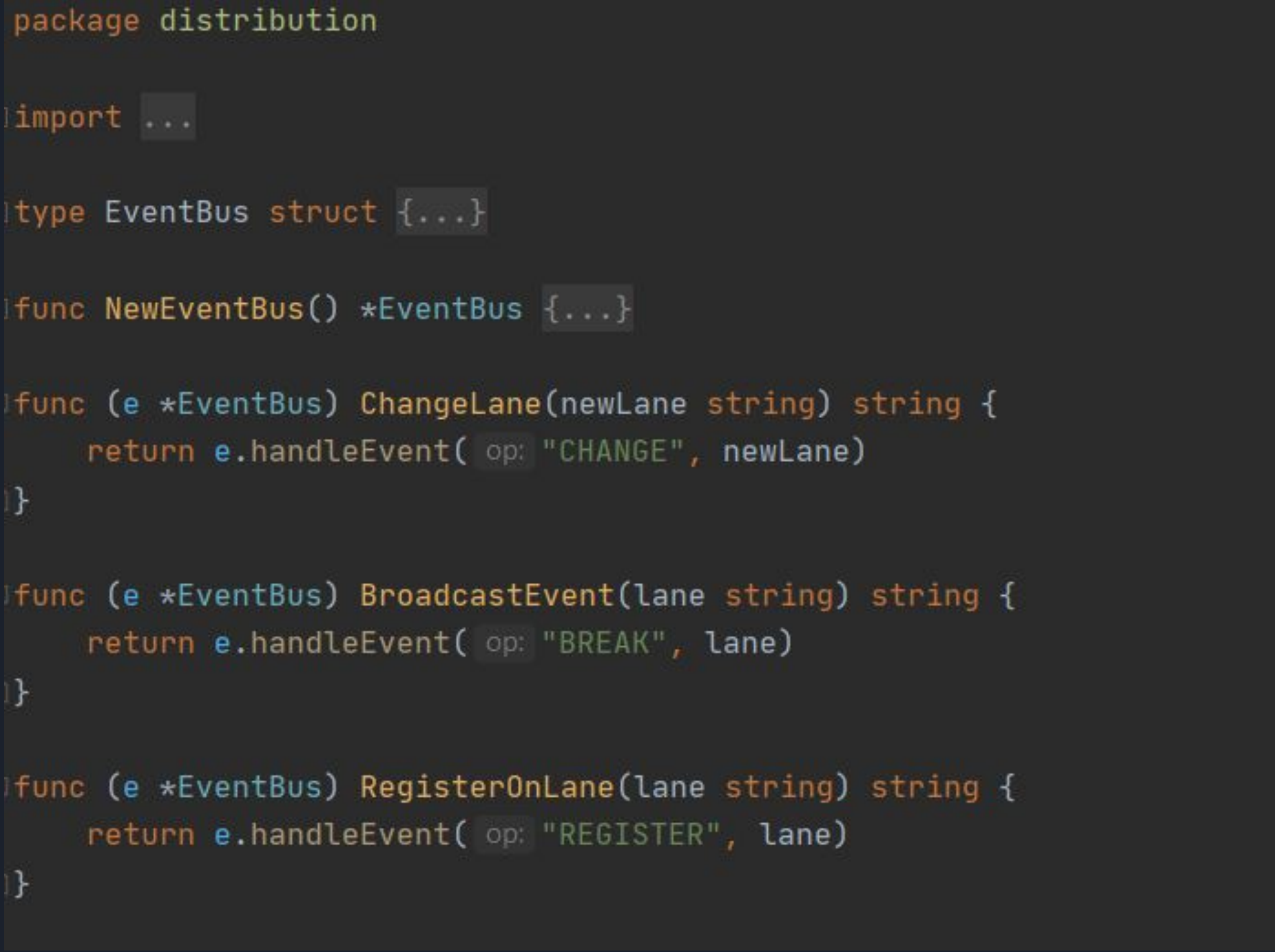
```go
func CreateLookupMessageFromLookupPacket(packet *Packet) *LookupMessage {
    if string(packet.Header) == "lookup" {
        var aorBody []byte
        var messageBody []byte
        var lastOne = false
        for i, b := range packet.Body {...}
        aor := &AOR{}
        message := &Message{}
        _ = json.Unmarshal(aorBody, aor)
        _ = json.Unmarshal(messageBody, message)
        return &LookupMessage{
            Message: message,
            AOR:     aor,
        }
    }
    return nil
}
```

Packet



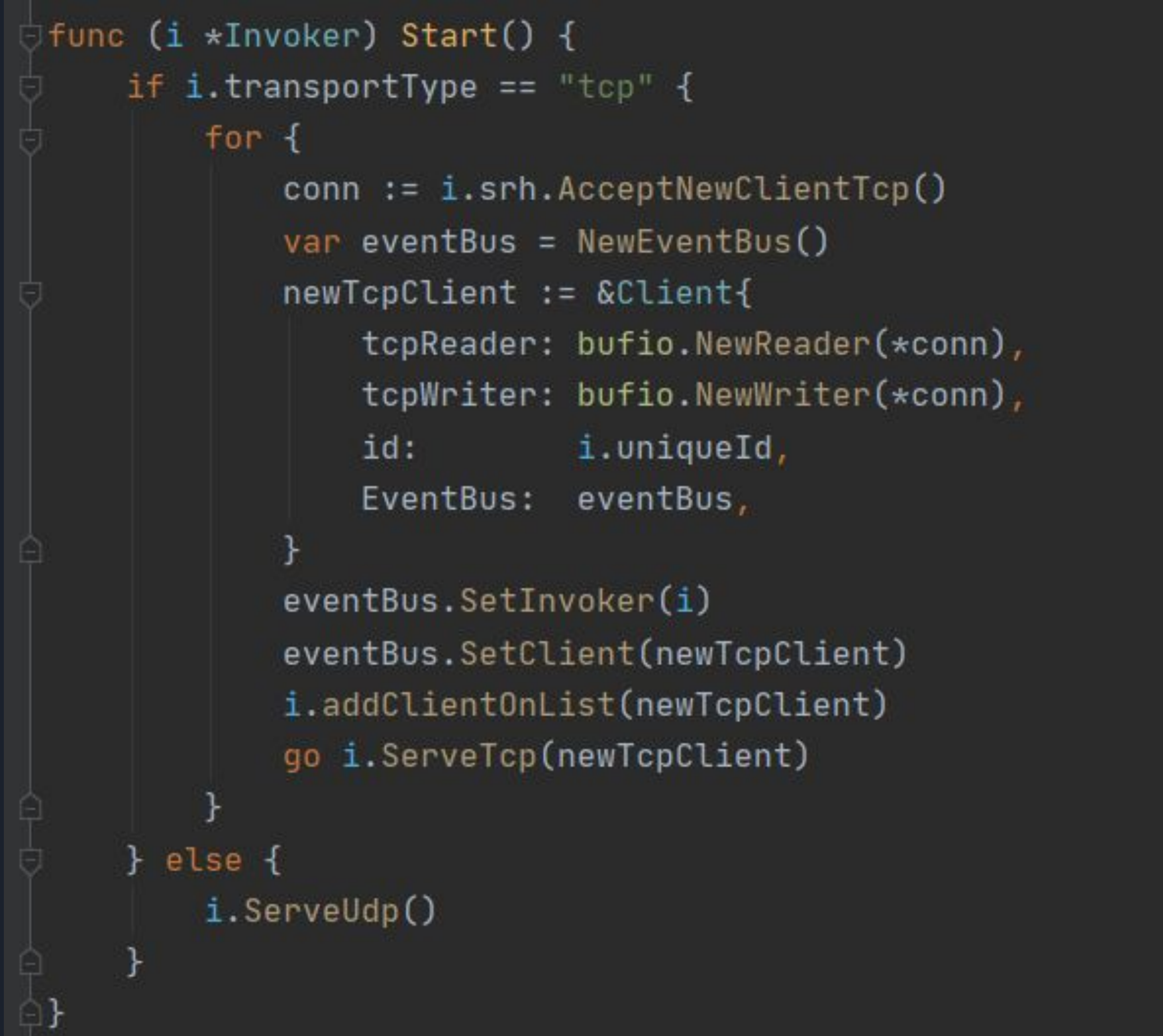
```
func NewLookUpReplyPacket(aor interface{}) *Packet {
    fmt.Println( a...: "Creating reply package")
    header := []byte("lookup")
    body, _ := json.Marshal(aor)
    return &Packet{
        Header: header,
        Body:   body,
    }
}
```

EventBus



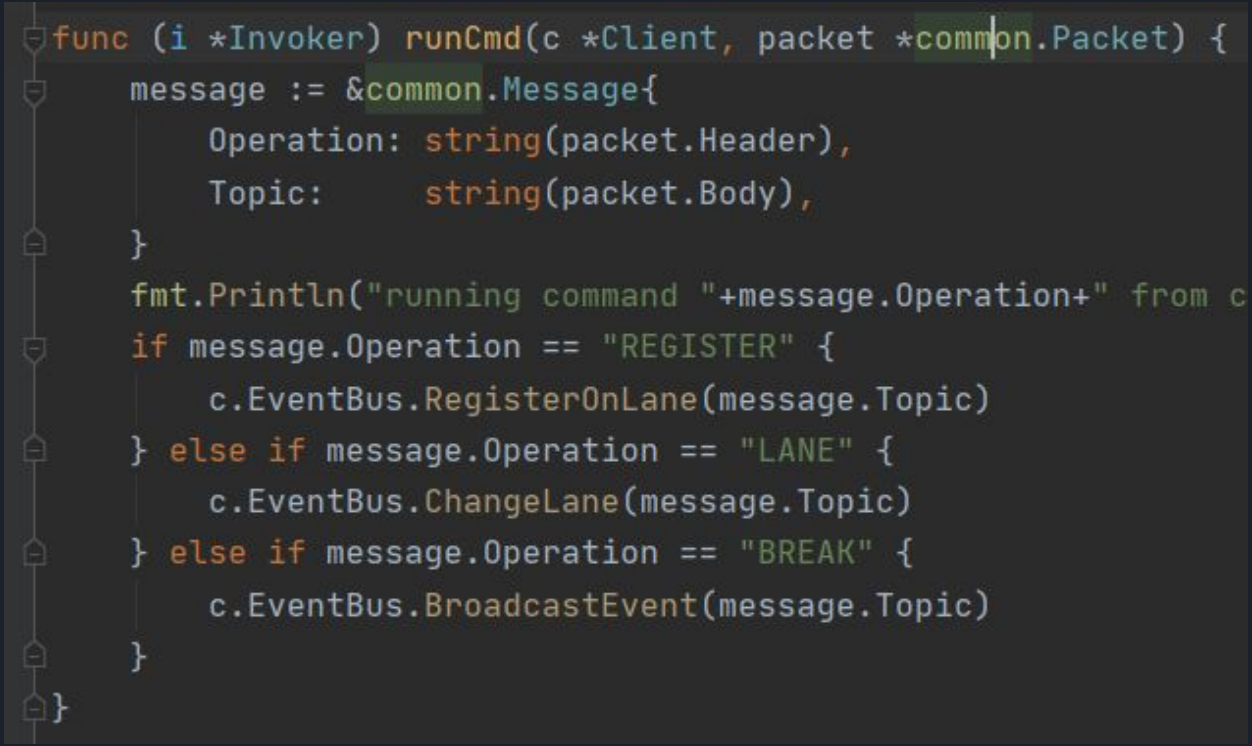
```
package distribution

import ...

type EventBus struct {...}

func NewEventBus() *EventBus {...}

func (e *EventBus) ChangeLane(newLane string) string {
    return e.handleEvent( op: "CHANGE", newLane)
}

func (e *EventBus) BroadcastEvent(lane string) string {
    return e.handleEvent( op: "BREAK", lane)
}

func (e *EventBus) RegisterOnLane(lane string) string {
    return e.handleEvent( op: "REGISTER", lane)
}
```

Invoker



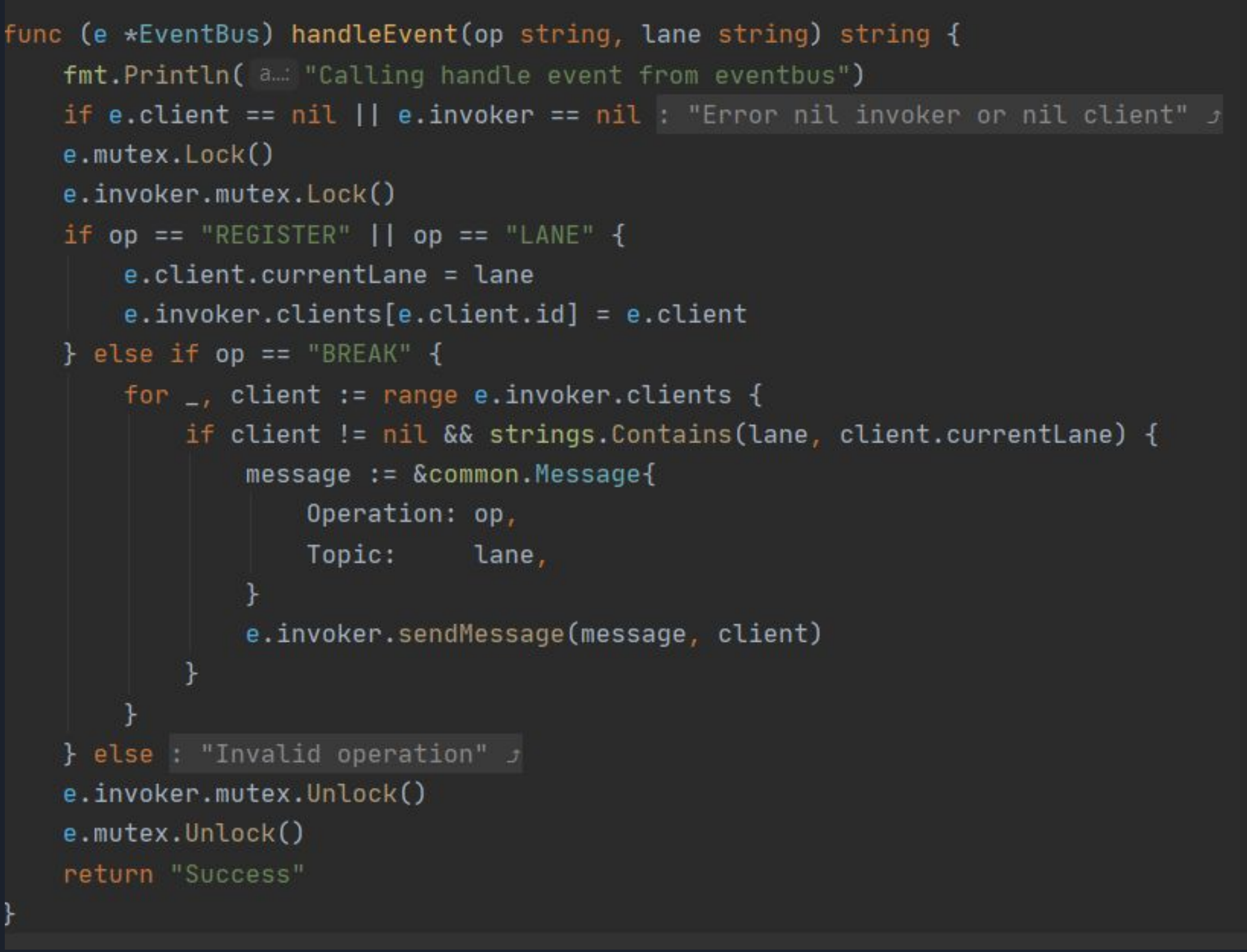
```go
func (i *Invoker) Start() {
    if i.transportType == "tcp" {
        for {
            conn := i.srh.AcceptNewClientTcp()
            var eventBus = NewEventBus()
            newTcpClient := &Client{
                tcpReader: bufio.NewReader(*conn),
                tcpWriter: bufio.NewWriter(*conn),
                id:        i.uniqueId,
                EventBus:  eventBus,
            }
            eventBus.SetInvoker(i)
            eventBus.SetClient(newTcpClient)
            i.addClientOnList(newTcpClient)
            go i.ServeTcp(newTcpClient)
        }
    } else {
        i.ServeUdp()
    }
}
```

Invoker



```
func (i *Invoker) runCmd(c *Client, packet *common.Packet) {
    message := &common.Message{
        Operation: string(packet.Header),
        Topic:     string(packet.Body),
    }
    fmt.Println("running command "+message.Operation+" from c
    if message.Operation == "REGISTER" {
        c.EventBus.RegisterOnLane(message.Topic)
    } else if message.Operation == "LANE" {
        c.EventBus.ChangeLane(message.Topic)
    } else if message.Operation == "BREAK" {
        c.EventBus.BroadcastEvent(message.Topic)
    }
}
```

EventBus



```
func (e *EventBus) handleEvent(op string, lane string) string {
    fmt.Println( a…: "Calling handle event from eventbus")
    if e.client == nil || e.invoker == nil : "Error nil invoker or nil client" ⤶
    e.mutex.Lock()
    e.invoker.mutex.Lock()
    if op == "REGISTER" || op == "LANE" {
        e.client.currentLane = lane
        e.invoker.clients[e.client.id] = e.client
    } else if op == "BREAK" {
        for _, client := range e.invoker.clients {
            if client != nil && strings.Contains(lane, client.currentLane) {
                message := &common.Message{
                    Operation: op,
                    Topic:     lane,
                }
                e.invoker.sendMessage(message, client)
            }
        }
    } else : "Invalid operation" ⤶
    e.invoker.mutex.Unlock()
    e.mutex.Unlock()
    return "Success"
}
```