

# Relatório da Atividade 2:

## Exclusão Mútua

Isabelle Ferreira de Oliveira  
CES-27 - Engenharia da Computação 2020  
Instituto Tecnológico de Aeronáutica (ITA)  
São José dos Campos, Brasil  
isabelle.ferreira3000@gmail.com

**Resumo**—Esse relatório documenta a implementação do algoritmo de Ricart-Agrawala, um algoritmo de exclusão mútua para sistemas distribuídos.

**Index Terms**—Algoritmo de Ricart-Agrawala, exclusão mútua, Relógio lógico vetorial, sistemas distribuídos

### I. IMPLEMENTAÇÃO

#### A. Recurso Compartilhado

A implementação do recurso compartilhado se tratou principalmente da utilização do código disponibilizado no roteiro do laboratório. Dentro do loop principal, foi colocada a função de servidor, cuja implementação era simplesmente imprimir na tela qualquer mensagem recebida pela porta :10001.

Abaixo encontra-se os códigos da função `main()` e da função `doServerJob()`.

```
func main() {
    Address, err := net.ResolveUDPAddr("udp",
        ":10001")
    Connection, err = net.ListenUDP("udp",
        Address)
    defer Connection.Close()

    for {
        doServerJob()
    }
}

func doServerJob() {
    buf := make([]byte, 1024)

    n, _, err := Connection.ReadFromUDP(buf)
    err = json.Unmarshal(buf[:n],
        &messageReceived)

    fmt.Println(messageReceived.Text)
}
```

A mensagem recebida era da forma de struct, com os atributos de id do processo que enviou a mensagem, o relógio lógico do processo que enviou a mensagem e, por fim, o texto a ser impresso no terminal.

Vale ressaltar também que, para esse código e todos os outros dessa atividade, sempre após a setagem da variável `err`, referente a um possível erro advindo de algumas funções,

também era chamada a função `CheckError(err)`, que imprime o erro e interrompe o processo caso houvesse algum erro.

#### B. Processo

A grosso modo, o processo foi implementado de forma bastante semelhante ao feito no Laboratório 1, como pode ser observado na `main()` abaixo.

Sobre as diferenças, a existência de estados ("RELEASED", "WANTED" e "HELD"), que determinam a validade de uma entrada pelo terminal do processo, além participam na priorização de permanência na CS (assim como o timestamp das requests de entrada na CS, e os IDs dos processos). Além disso, mudanças nos comportamentos do servidor, além de mudanças nas mensagens trocadas entre os processos.

Conforme se pode ver na `main()` apresentada abaixo, ao receber uma mensagem pelo terminal, o processo verifica sua validade e, caso seja válida, passa para o estado "WANTED", enviando requests de entrada na CS para os demais processos. A partir daí, o processo em questão espera pelas replies dos demais processos, entrando na CS quando receber todas. Após sair da CS, o processo responde as requests dos demais processos que ele tiver adicionado a fila.

```
func main() {
    initConnections()
    setState("RELEASED")

    defer ServerConn.Close()
    for i := 0; i < nPorts; i++ {
        defer ClientsConn[i].Close()
    }

    go readInput(ch)

    for {
        go doServerJob()

        select {
        case textReceived, valid := <-ch:
            if valid {
                if myState == "WANTED" || myState ==
                    "HELD" {
                    fmt.Println(textReceived,
                        "invalido")
                } else {
                    if textReceived != myIDString {
                        messageSent.Text = textReceived
                    }
                }
            }
        }
    }
}
```

```

        setState("WANTED")
        myTimestamp = logicalClock
        request.Timestamp = myTimestamp

        for otherID := 1; otherID <=
            nPorts; otherID++ {
            if otherID != myID {
                go doClientJob(request,
                    otherID)
            }
        }
        go waitReplies()
    } else {
        // updating my clock
        logicalClock++
    }
}
} else {
    fmt.Println("Channel closed!")
}
default:
    time.Sleep(time.Second * 1)
}
}

```

Como servidor, ao receber uma mensagem de outros processos, o processo atualiza seu relógio lógico e verifica o tipo dessa mensagem, se é "request" ou "reply". Essa mensagem trata-se de uma struct, contendo os atributos de tipo ("request" ou "reply"), além de o timestamp (para o caso de request), o ID do processo que enviou e seu relógio lógico.

Caso receba um reply de que pode entrar na CS, um contador de replies é acrescido. Já caso a mensagem seja um request, é necessário verificar as condições de prioridade para responder esse request, ou adicioná-lo a uma fila com os demais requests a se responder. Enviar uma reply também incrementa o relógio lógico do processo.

Segue a seguir o código da função doServerJob().

```

func doServerJob() {
    buf := make([]byte, 1024)

    n, _, err := ServerConn.ReadFromUDP(buf)
    var message RequestReplyStruct
    err = json.Unmarshal(buf[:n], &message)

    msgType := message.Type
    msgLogicalClock := message.LogicalClock
    msgTimestamp := message.Timestamp

    // updating clocks
    logicalClock = max(msgLogicalClock,
        logicalClock) + 1

    if msgType == "request" {
        msgId := message.Id

        if myState == "HELD" ||
            ( myState == "WANTED" && (
                msgTimestamp < myTimestamp ||
                ( msgTimestamp == myTimestamp &&
                    msgId < myID ))) {

```

```

            requestsQueue = append(requestsQueue,
                msgId)
        } else {
            // updating clocks
            logicalClock++
            reply.LogicalClock = logicalClock

            jsonReply, err := json.Marshal(reply)
            _, err =
                CliConn[msgId-1].Write(jsonReply)
        }
    } else if msgType == "reply" {
        nReplies++
    }
}

```

Por fim, esperar as replies dos demais processos se trata de uma thread presa em loop com a condição de o contador de replies chegar ao total esperado. Após isso, o processo passa para o estado "HELD", adentra a CS e a usa, posteriormente saindo da CS e passando para o estado "RELEASED". Como explicado anteriormente, dentro da CS, o processo envia uma mensagem ao SharedResource com o texto digitado no terminal do processo e dorme por alguns segundos; após sair da CS, o processo responde aos requests restantes que estiverem enfileirados.

Os códigos de waitReplies() e useCS() estão apresentados a seguir.

```

func waitReplies() {
    for nReplies != nPorts-1 {}
    nReplies = 0

    setState("HELD")
    useCS()
    setState("RELEASED")

    // reply requests
    for _, element := range requestsQueue {
        logicalClock++
        reply.LogicalClock = logicalClock
        jsonReply, err := json.Marshal(reply)
        _, err =
            ClientsConn[element-1].Write(jsonReply)
    }
    requestsQueue = make([]int, 0)
}

```

```

func useCS() {
    fmt.Println("Entrei na CS")
    messageSent.LogicalClock = logicalClock

    jsonMessage, err :=
        json.Marshal(messageSent)
    _, err =
        SharedResourceConn.Write(jsonMessage)

    time.Sleep(time.Second * 10)
    fmt.Println("Sai da CS")
}

```

Caso seja necessário, pode-se também consultar o código enviado como anexo a essa atividade.

## II. RESULTADOS E CONCLUSÕES

### A. Teste 1

Este foi o caso com um processo solicitando a CS e, depois que ele liberasse, outro processo solicitando a CS, sugerido no roteiro do laboratório. O esquema do resultado esperado foi apresentado na Figura 1.

Na Figura 1, para P1, as setas azuis representam requests e as verdes, replies; para P3, essas setas são rosas e cinzas, respectivamente. Nos requests, a mensagem enviada é da forma "relógio lógico, < timestamp, id >"; já no reply, a forma é "relógio lógico, 'reply'". Além disso, a linha amarela representa o processo no estado WANTED; e a linha vermelha, no estado HELD, ou seja, na CS.

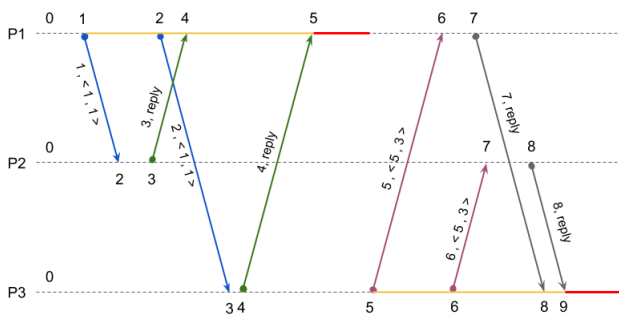


Figura 1. Funcionamento esperado para a tarefa com 3 processos.

Foi simulada essa situação acima com o código implementado no laboratório, tendo os resultados apresentados nas Figuras de 2 a 5. Na simulação, ao invés de o P3 fazer a requisição, é o P2 a faz, mas isso não prejudica o teste.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-
$ go run Process.go 1 :10006 :10007 :10008
oi
Entrei na CS
Sai da CS
```

Figura 2. Exemplo do funcionamento da tarefa com 3 processos. Tela do processo 1.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-
$ go run Process.go 2 :10006 :10007 :10008
xau
Entrei na CS
Sai da CS
```

Figura 3. Exemplo do funcionamento da tarefa com 3 processos. Tela do processo 2.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-
$ go run Process.go 3 :10006 :10007 :10008
```

Figura 4. Exemplo do funcionamento da tarefa com 3 processos. Tela do processo 3.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-
$ go run SharedResource.go
oi
xau
```

Figura 5. Exemplo do funcionamento da tarefa com 3 processos. Tela do SharedResource.

A fim de entender melhor cada estágio do funcionamento, foram realizados os mesmos testes novamente, agora com prints de debug. Dessa forma, esses resultados estão apresentados nas Figuras de 6 a 9.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-
$ go run Process.go 1 :10006 :10007 :10008
Estado: RELEASED
oi
Estado: WANTED
Multicast request to all processes
logicalClock atualizado: 1
Request enviado: 1 , < 0 , 1 >
Esperando N-1 respostas
logicalClock atualizado: 2
Request enviado: 2 , < 0 , 1 >
Reply recebido: 3 reply
logicalClock atualizado: 4
Reply recebido: 4 reply
logicalClock atualizado: 5
Estado: HELD
Entrei na CS
Request recebido: 6 , < 4 , 3 >
logicalClock atualizado: 7
Sai da CS
Estado: RELEASED
Reply enviado: 8 reply
```

Figura 6. Exemplo do funcionamento da tarefa com 3 processos. Tela do processo 1.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata
└─$ go run Process.go 2 :10006 :10007 :10008
Estado: RELEASED
Request recebido: 1 , < 0 , 1 >
logicalClock atualizado: 2
Reply enviado: 3 reply
logicalClock atualizado: 3
Request recebido: 6 , < 4 , 3 >
logicalClock atualizado: 7
Reply enviado: 8 reply
logicalClock atualizado: 8

```

Figura 7. Exemplo do funcionamento da tarefa com 3 processos. Tela do processo 2.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana
└─$ go run Process.go 3 :10006 :10007 :10008
Estado: RELEASED
xauRequest recebido: 2 , < 0 , 1 >
logicalClock atualizado: 3
Reply enviado: 4 reply
logicalClock atualizado: 4

Estado: WANTED
Multicast request to all processes
Esperando N-1 respostas
logicalClock atualizado: 6
Request enviado: 6 , < 4 , 3 >
logicalClock atualizado: 5
Request enviado: 6 , < 4 , 3 >
Reply recebido: 8 reply
logicalClock atualizado: 9
Reply recebido: 8 reply
logicalClock atualizado: 10
Estado: HELD
Entrei na CS
Sai da CS
Estado: RELEASED

```

Figura 8. Exemplo do funcionamento da tarefa com 3 processos. Tela do processo 3.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-
└─$ go run SharedResource.go
Received {1 5 oi}
oi
Received {3 10 xau}
xau

```

Figura 9. Exemplo do funcionamento da tarefa com 3 processos. Tela do SharedResource.

Como esses resultados foram condizentes com os resultados esperados, leva-se a perceber que a implementação da Tarefa foi feita corretamente. Mas, antes de concluir-se algo, foi-se realizado um segundo teste.

## B. Teste 2

Este foi o caso com processos solicitando a CS "simultaneamente", sugerido no roteiro do laboratório. O esquema do resultado esperado foi apresentado na Figura 10.

Análogo ao teste 1, na Figura 10, para P1, as setas azuis representam requests e as verdes, replies; e para P4, essa setas são rosas e cinzas, respectivamente. Nos requests, a mensagem enviada também é da forma "relógio lógico, < timestamp, id >"; assim como no reply, que a forma também é "relógio lógico, 'reply'". Além disso, nesse caso também a linha amarela representa o processo no estado WANTED; e a linha vermelha, no estado HELD, ou seja, na CS.

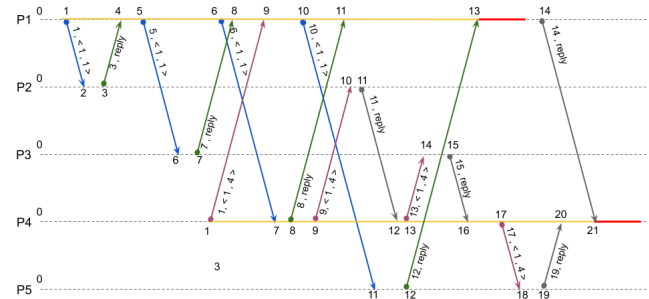


Figura 10. Funcionamento esperado para a tarefa para 5 processos.

Foi simulada essa situação acima com o código implementado no laboratório, tendo os resultados apresentados nas Figuras de 11 a 16. Durante o período na CS, também foi digitado mensagens no terminal dos processos, para que se verificasse o funcionamento da validação da mensagem. Assim, para essas mensagens, foi dado o feedback de mensagem inválida.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-CES-27/Lab2 <master>
└─$ go run Process.go 1 :10004 :10003 :10002 :10005 :10006
oi
Entrei na CS
a
a invalido
Sai da CS

```

Figura 11. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 1.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-CES-27/Lab2 <master>
└─$ go run Process.go 2 :10004 :10003 :10002 :10005 :10006

```

Figura 12. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 2.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-CES-27/Lab2 «master*»  
└─$ go run Process.go 3 :10004 :10003 :10002 :10005 :10006
```

Figura 13. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 3.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-CES-27/Lab2 «master*»  
└─$ go run Process.go 4 :10004 :10003 :10002 :10005 :10006  
xau  
Entrei na CS  
a  
a invalido  
4  
4 invalido  
Sai da CS
```

Figura 14. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 4.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-CES-27/Lab2 «master*»  
└─$ go run Process.go 5 :10004 :10003 :10002 :10005 :10006
```

Figura 15. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 5.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-CES-27/Lab2 «master*»  
└─$ go run SharedResource.go  
oi  
xau
```

Figura 16. Exemplo do funcionamento da tarefa com 5 processos. Tela do SharedResource.

A fim de entender melhor cada estágio do funcionamento, foram realizados os mesmos testes novamente, agora com prints de debug. Dessa forma, esses resultados estão apresentados nas Figuras de 17 a 22. Como é difícil simular com exatidão os instantes dos envios de request dos processos simultaneamente, foi feito da seguinte forma: o processo P4 entra no estado WANTED durante o estado HELD do processo P1. Isso é um pouco diferente do apresentado na Figura 10, mas não prejudica os testes. Contudo, isso altera um pouco a ordem dos envios de mensagens, e, consequentemente, os valores relógios lógicos.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-CES-27/Lab2 «master*»  
└─$ go run Process.go 1 :10004 :10003 :10002 :10005 :10006  
Estado: RELEASED  
oi  
Estado: WANTED  
Multicast request to all processes  
Esperando N-1 respostas  
logicalClock atualizado: 1  
Request enviado: 2 , < 0 , 1 >  
logicalClock atualizado: 2  
Request enviado: 2 , < 0 , 1 >  
logicalClock atualizado: 3  
Request enviado: 3 , < 0 , 1 >  
logicalClock atualizado: 4  
Request enviado: 4 , < 0 , 1 >  
Reply recebido: 5 reply  
logicalClock atualizado: 6  
Reply recebido: 6 reply  
logicalClock atualizado: 7  
Reply recebido: 4 reply  
logicalClock atualizado: 8  
Reply recebido: 4 reply  
logicalClock atualizado: 9  
Estado: HELD  
Entrei na CS  
Request recebido: 6 , < 4 , 4 >  
logicalClock atualizado: 10  
Sai da CS  
Estado: RELEASED  
Reply enviado: 11 reply
```

Figura 17. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 1.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-CES-27/Lab2 «master*»  
└─$ go run Process.go 2 :10004 :10003 :10002 :10005 :10006  
Estado: RELEASED  
Request recebido: 3 , < 0 , 1 >  
logicalClock atualizado: 4  
Reply enviado: 5 reply  
logicalClock atualizado: 5  
Request recebido: 8 , < 4 , 4 >  
logicalClock atualizado: 9  
Reply enviado: 10 reply  
logicalClock atualizado: 10
```

Figura 18. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 2.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-CES-27/Lab2 «master*»  
└─$ go run Process.go 3 :10004 :10003 :10002 :10005 :10006  
Estado: RELEASED  
Request recebido: 4 , < 0 , 1 >  
logicalClock atualizado: 5  
Reply enviado: 6 reply  
logicalClock atualizado: 6  
Request recebido: 7 , < 4 , 4 >  
logicalClock atualizado: 8  
Reply enviado: 9 reply  
logicalClock atualizado: 9
```

Figura 19. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 3.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana
└─$ go run Process.go 4 :10004 :10003 :10002 :10005 :10006
Estado: RELEASED
xauRequest recebido: 2 , < 0 , 1 >
logicalClock atualizado: 3
Reply enviado: 4 reply
logicalClock atualizado: 4

Estado: WANTED
Multicast request to all processes
logicalClock atualizado: 6
Request enviado: 6 , < 4 , 4 >
logicalClock atualizado: 7
Esperando N-1 respostas
logicalClock atualizado: 8
Request enviado: 8 , < 4 , 4 >
logicalClock atualizado: 5
Request enviado: 8 , < 4 , 4 >
Reply recebido: 10 reply
logicalClock atualizado: 11
Request enviado: 7 , < 4 , 4 >
Reply recebido: 10 reply
logicalClock atualizado: 12
Reply recebido: 9 reply
logicalClock atualizado: 13
Reply recebido: 11 reply
logicalClock atualizado: 14
Estado: HELD
Entrei na CS
Sai da CS
Estado: RELEASED

```

Figura 20. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 4.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-C
└─$ go run Process.go 5 :10004 :10003 :10002 :10005 :10006
Estado: RELEASED
Request recebido: 2 , < 0 , 1 >
logicalClock atualizado: 3
Reply enviado: 4 reply
logicalClock atualizado: 4
Request recebido: 8 , < 4 , 4 >
logicalClock atualizado: 9
Reply enviado: 10 reply
logicalClock atualizado: 10

```

Figura 21. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 5.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-CES-27/L
└─$ go run SharedResource.go
Received {1 9 oi}
oi
Received {4 14 xau}
xau

```

Figura 22. Exemplo do funcionamento da tarefa com 5 processos. Tela do SharedResource.

Como esses resultados também foram condizentes com os resultados esperados, conclui-se que a implementação da Tarefa foi feita corretamente.