

**CES-27/2018: Processamento Distribuído**  
**CTA – ITA – IEC**  
**Prof.a Juliana Bezerra e Prof Vitor Curtis**

**Atividade 2: Logical Clock**  
**Aluno: Gustavo Nahum Alvarez Ferreira**

### Tarefa 1

No programa, há dois códigos sendo executados: o do cliente e o do servidor.

No do cliente, a função *ResolveUDPAddr* retorna o endereço de ponto final UDP, obtendo-se assim o endereço do servidor, *ServerAddr*, e do cliente, *LocalAddr*. A seguir, a função *DialUDP* faz a discagem do *LocalAddr* para o *ServerAddr*.

A chamada *defer Conn.Close()* fecha a conexão assim que a main for terminada, isto é, quando o executável for encerrado.

No loop principal, o inteiro *i* é convertido para uma mensagem em formato string que, por sua vez, é convertida em um buffer em formato slice de bytes. Esse buffer é, então, enviado através da variável de conexão. A cada iteração desse loop principal, o inteiro *i* é acrescido de 1, para verificar-se o avanço dos pacotes enviados.

Já no código do servidor, este chama a função *ResolveUDPAddr* para receber seu endereço de ponto final UDP, que fica armazenada na variável *ServerAddr*.

A seguir, cria a variável *ServerConn* a partir da função *ListenUDP*, responsável por escutar no seu endereço, até que receba algum pacote na conexão.

Então cria um buffer (que é novamente um slice de bytes) com tamanho 1024, que irá receber os pacotes enviados pelo cliente na conexão UDP.

Assim que recebe algum pacote, imprime o que está no buffer.

O funcionamento do programa cliente-servidor usando conexão UDP foi testado com o cliente ligado em um terminal, e o servidor em outro terminal.

e2/Tarefa1\$ ./client 1 write udp 127.0.0.1:47613->127.0.0.1:10001: write: connection refused 3 write udp 127.0.0.1:47613->127.0.0.1:10001: write: connection refused 5 write udp 127.0.0.1:47613->127.0.0.1:10001: write: connection refused 7 write udp 127.0.0.1:47613->127.0.0.1:10001: write: connection refused 23 write udp 127.0.0.1:47613->127.0.0.1:10001: write: connection refused 25 write udp 127.0.0.1:47613->127.0.0.1:10001: write: connection refused 27 write udp 127.0.0.1:47613->127.0.0.1:10001: write: connection refused 29 write udp 127.0.0.1:47613->127.0.0.1:10001: write: connection refused ^C	e2/Tarefa1\$ ./server Received 8 from 127.0.0.1:47613 Received 9 from 127.0.0.1:47613 Received 10 from 127.0.0.1:47613 Received 11 from 127.0.0.1:47613 Received 12 from 127.0.0.1:47613 Received 13 from 127.0.0.1:47613 Received 14 from 127.0.0.1:47613 Received 15 from 127.0.0.1:47613 Received 16 from 127.0.0.1:47613 Received 17 from 127.0.0.1:47613 Received 18 from 127.0.0.1:47613 Received 19 from 127.0.0.1:47613 Received 20 from 127.0.0.1:47613 Received 21 from 127.0.0.1:47613 ^C gustavo-nahum@debian-stretch:~/Documentos/ITA/Juliana/Atividades/Repo-Github/CES-27/Atividad
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figura 1. Teste de comunicação UDP entre o cliente (à esquerda) e servidor (à direita).

Quando o cliente estava ligado, mas o servidor não, o terminal do cliente imprimia uma mensagem de erro, alertando que a conexão entre a porta do cliente e a porta do servidor (10001) foi recusada.

Já quando o servidor estava ligado, porém o cliente não, o terminal do servidor não imprime nenhuma mensagem, demonstrando que o servidor apenas permanece escutando sua porta, porém sem estabelecer comunicação com o cliente.

Por fim, quando ambos estão ligados, o terminal do servidor imprime as mensagens que foram recebidas da comunicação com o cliente, isto é, a sequência de números inteiros em ordem crescente, acompanhada do endereço em que está situado o cliente quando a comunicação foi estabelecida.

Nota-se, em particular, que quando o cliente está ligado, mas o servidor não, o cliente não imprime os pacotes pares que foram perdidos, mas apenas os ímpares. Possivelmente, isso se justifica pelo motivo de que o envio de mensagens não é instantâneo através da conexão, e a escrita de um buffer em *Write* aguarda até que seja recebida pelo servidor; quando o servidor está desligado, o pacote não consegue ser recebido, e fica aguardando; assim, quando chega na iteração seguinte do loop principal, tenta-se enviar nova mensagem sem ter concluído o envio da mensagem anterior, e só então identifica-se o erro. Dessa forma, como inicialmente *i* era 0, ao chegar em 1 ocorre colisão e imprime-se erro em 1; a seguir, *i* passa a ser 2 mas não consegue enviar sua mensagem, e quando chega o pacote 3 ocorre a colisão e imprime-se o erro em 3, etc.

Código *client.go*:

```
package main

import (
    "fmt"
    "net"
    "time"
    "strconv"
)

func CheckError(err error) {
    if err != nil {
        fmt.Println("Error: ", err)
    }
}

func main() {
    ServerAddr,err := net.ResolveUDPAddr("udp","127.0.0.1:10001")
    CheckError(err)

    LocalAddr, err := net.ResolveUDPAddr("udp", "127.0.0.1:0")
    CheckError(err)

    Conn, err := net.DialUDP("udp", LocalAddr, ServerAddr)
    CheckError(err)

    defer Conn.Close()
    i := 0

    for {
        msg := strconv.Itoa(i)
        i++
        buf := []byte(msg)
        _,err := Conn.Write(buf)
        if err != nil {
            fmt.Println(msg, err)
        }
    }
}
```

```

        }
        time.Sleep(time.Second * 5)
    }
}

```

Código server.go:

```

package main

import (
    "fmt"
    "net"
    "os"
)

/* A Simple function to verify error */
func CheckError(err error) {
    if err != nil {
        fmt.Println("Error: ", err)
        os.Exit(0)
    }
}

func main() {
    /* Lets prepare a address at any address at port 10001*/
    ServerAddr,err := net.ResolveUDPAddr("udp","::10001")
    CheckError(err)

    /* Now listen at selected port */
    ServerConn, err := net.ListenUDP("udp", ServerAddr)
    CheckError(err)
    defer ServerConn.Close()

    buf := make([]byte, 1024)

    for {
        n,addr,err := ServerConn.ReadFromUDP(buf)
        fmt.Println("Received ",string(buf[0:n]), " from ",addr)

        if err != nil {
            fmt.Println("Error: ",err)
        }
    }
}

```

## Tarefa 2

Ao juntar em um só arquivo *Process.go* os códigos do cliente e do servidor, um único processo passou a ser capaz de enviar e de receber mensagens. Assim, foi possível ter vários processos se comunicando.

Para isso, foi necessário criar um slice de conexões de clientes, com tamanho no mínimo igual à quantidade de servidores com os quais cada processo deverá se comunicar. Isso foi feito por meio do uso da biblioteca *os*, que armazenava os argumentos passados no momento da execução do processo: eles puderam ser recuperados acessando o array *os.Args[i]*.

No resto, o código seguiu essencialmente a lógica da tarefa 1, bastando apenas organizar nas funções *doServerJob* e *doClientJob* as tarefas específicas dos servidores e clientes, respectivamente.

```

e2/Tarefa2$ ./Process 10004 10003
1 write udp 127.0.0.1:51103->127.0.0.1:10003:
write: connection refused
3 write udp 127.0.0.1:51103->127.0.0.1:10003:
write: connection refused
5 write udp 127.0.0.1:51103->127.0.0.1:10003:
write: connection refused
7 write udp 127.0.0.1:51103->127.0.0.1:10003:
write: connection refused
Received 0 from 127.0.0.1:50739
Received 1 from 127.0.0.1:50739
Received 2 from 127.0.0.1:50739
Received 3 from 127.0.0.1:50739
Received 4 from 127.0.0.1:50739
Received 5 from 127.0.0.1:50739
Received 6 from 127.0.0.1:50739
Received 7 from 127.0.0.1:50739
Received 8 from 127.0.0.1:50739
Received 9 from 127.0.0.1:50739
Received 10 from 127.0.0.1:50739
Received 11 from 127.0.0.1:50739
^C
e2/Tarefa2$ ./Process 10003 10004
Received 8 from 127.0.0.1:51103
Received 9 from 127.0.0.1:51103
Received 10 from 127.0.0.1:51103
Received 11 from 127.0.0.1:51103
Received 12 from 127.0.0.1:51103
Received 13 from 127.0.0.1:51103
Received 14 from 127.0.0.1:51103
Received 15 from 127.0.0.1:51103
Received 16 from 127.0.0.1:51103
Received 17 from 127.0.0.1:51103
Received 18 from 127.0.0.1:51103
Received 19 from 127.0.0.1:51103
13 write udp 127.0.0.1:50739->127.0.0.1:10004:
    write: connection refused
15 write udp 127.0.0.1:50739->127.0.0.1:10004:
    write: connection refused
17 write udp 127.0.0.1:50739->127.0.0.1:10004:
    write: connection refused
^C
gustavo-nahum@debian-stretch:~/Documentos/ITA/Juliana/Atividades/Repo-Github/CES-27/Atividad

```

Figura 2. Teste de conexão UDP entre dois processos: um com o servidor na porta 10004 (à esquerda) e o outro com o servidor na porta 10003 (à direita).

Num primeiro teste (figura 2), fez-se a comunicação entre dois processos, um deles com o servidor recebendo a porta 10004, e no outro o servidor recebendo a porta 10003. Ao rodar o sistema nessa configuração, verificou-se que a comunicação foi estabelecida com êxito: cada processo continuou recebendo números inteiros sequenciais do outro processo. Contudo, a sequência de inteiros que eles recebiam diferia entre si: o processo que foi iniciado no terminal mais tarde recebeu, como primeira mensagem, não o número “0”, mas o “8”, porque as demais mensagens não foram recebidas; em contrapartida, este processo que iniciou mais tarde começou a transmissão de suas mensagens quando o outro já estava ativo, portanto todas as suas mensagens foram entregues com sucesso, e o primeiro processo imprimiu inteiros a partir do número “0”.

```

write: connection refused
Received 1 from 127.0.0.1:55355
Received 2 from 127.0.0.1:55355
7 write udp 127.0.0.1:47528->127.0.0.1:10004:
write: connection refused
Received 3 from 127.0.0.1:55355
Received 6 from 127.0.0.1:60788
Received 4 from 127.0.0.1:55355
Received 1 from 127.0.0.1:60788
Received 5 from 127.0.0.1:55355
Received 2 from 127.0.0.1:60788
Received 6 from 127.0.0.1:55355
Received 3 from 127.0.0.1:60788
Received 7 from 127.0.0.1:55355
Received 4 from 127.0.0.1:60788
Received 8 from 127.0.0.1:55355
Received 5 from 127.0.0.1:60788
Received 9 from 127.0.0.1:55355
Received 6 from 127.0.0.1:60788
Received 10 from 127.0.0.1:55355
^C
gustavo-nahum@debian-stretch:~/Documentos/ITA/Juliana/Atividades/Repo-Github/CES-27/Atividad
e2/Tarefa2$ ./Process 10003 10002 10004
1 write udp 127.0.0.1:58746->127.0.0.1:10004:
write: connection refused
Received 6 from 127.0.0.1:34756
Received 7 from 127.0.0.1:34756
3 write udp 127.0.0.1:58746->127.0.0.1:10004:
write: connection refused
Received 0 from 127.0.0.1:51780
Received 8 from 127.0.0.1:34756
Received 1 from 127.0.0.1:51780
Received 2 from 127.0.0.1:34756
Received 9 from 127.0.0.1:34756
Received 2 from 127.0.0.1:51780
Received 10 from 127.0.0.1:34756
Received 3 from 127.0.0.1:51780
Received 11 from 127.0.0.1:34756
Received 4 from 127.0.0.1:51780
Received 12 from 127.0.0.1:34756
Received 5 from 127.0.0.1:51780
Received 13 from 127.0.0.1:34756
Received 9 from 127.0.0.1:58746
11 write udp 127.0.0.1:60788->127.0.0.1:10002: write: connection refused
Received 14 from 127.0.0.1:47528
Received 10 from 127.0.0.1:58746
13 write udp 127.0.0.1:60788->127.0.0.1:10002: write: connection refused
^C
gustavo-nahum@debian-stretch:~/Documentos/ITA/Juliana/Atividades/Repo-Github/CES-27/Atividad

```

Figura 3. Teste de comunicação UDP entre três programas: um com o servidor na porta 10002 (à esquerda), outro com o servidor na porta 10003 (ao centro) e o outro com o servidor na porta 10004 (à direita).

No segundo teste (figura 3), fez-se a comunicação entre três processos, um com o servidor recebendo a porta 10002, outro com servidor na porta 10003 e o último com servidor na porta 10004. Novamente, pôde-se verificar que algumas mensagens começaram a ser entregues corretamente a partir do momento em que os dois primeiros processos foram ligados. Quando, por fim, o terceiro foi ligado, não houve mais mensagens de erro impressas, e todos estavam recebendo as mensagens que eram mandadas pelos outros dois processos. Além disso, verificou-se que cada processo recebia o mesmo número duas vezes (pois havia dois processos enviando esses mesmos números).

Código *Process2.go*:

```
package main

import (
    "fmt"
    "net"
    "os"
    "strconv"
    "time"
)

var err string
var myPort string
var nServers int
var CliConn []*net.UDPConn
var ServConn *net.UDPConn

func CheckError(err error) {
    if err != nil {
        fmt.Println("Error: ", err)
    }
}

func PrintError(err error) {
    if err != nil {
        fmt.Println("Error: ", err)
    }
}

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

    n,addr,err := ServConn.ReadFromUDP(buf)
    fmt.Println("Received ",string(buf[0:n]), " from ",addr)

    if err != nil {
        fmt.Println("Error: ",err)
    }
}

func doClientJob(otherProcess int, i int) {
    msg := strconv.Itoa(i)
    i++
    buf := []byte(msg)
    _,err := CliConn[otherProcess].Write(buf)
    if err != nil {
        fmt.Println(msg, err)
    }
    time.Sleep(time.Second * 1)
}

func initConnections() {
    myPort = os.Args[1]
    nServers = len(os.Args) - 2

    CliConn = make([]*net.UDPConn, nServers)

    ServerAddr, err := net.ResolveUDPAddr("udp", ":" + myPort)
    CheckError(err)
    ServConn, err = net.ListenUDP("udp", ServerAddr)
    CheckError(err)
```

```

    LocalAddr, err := net.ResolveUDPAddr("udp", "127.0.0.1:0")
    CheckError(err)
    for i:=0; i < nServers; i++ {
        otherServerAddr,err := net.ResolveUDPAddr("udp","127.0.0.1:" + os.Args[i + 2])
        CheckError(err)

        CliConn[i], err = net.DialUDP("udp", LocalAddr, otherServerAddr)
        CheckError(err)
    }
}

func main() {
    initConnections()
    defer ServConn.Close()
    for i := 0; i < nServers; i++ {
        defer CliConn[i].Close()
    }

    i := 0
    for {
        go doServerJob()
        for j := 0; j < nServers; j++ {
            go doClientJob(j, i)
        }
        time.Sleep(time.Second * 1)
        i++
    }
}

```

## Tarefa 3

Nessa tarefa, a comunicação UDP entre os processos foi mantida. Adicionalmente, foi incluída a interação com o terminal. Dessa forma, a inserção de caracteres no terminal era recebida e, em seguida, impressa novamente no próprio terminal.

Para isso, criou-se um canal de strings e acrescentou-se a função *readInput*, que por meio da chamada de *bufio.NewReader(os.Stdin)* recebia o que era digitado no terminal; essa entrada era, então, inserida no canal criado. Toda vez que o loop principal na main identificava algum conteúdo inserido no canal, imprimia novamente no terminal aquilo que havia sido digitado.

Com as alterações feitas no sistema, os processos foram capazes de trocar mensagens entre si como antes, porém além disso também passaram a reconhecer entradas do usuário no terminal.

Dessa forma, ao inserir caracteres no terminal de cada processo, o terminal imprimiu as entradas que recebeu do teclado, conforme era desejado. Esse comportamento não atrapalhou a impressão das mensagens que eram trocadas entre os processos, ou seja, as mensagens trocadas entre eles também foram impressas.

Esse comportamento pode ser verificado na figura 4: o terminal da esquerda foi o primeiro a ser iniciado, e alguns pacotes em sua transmissão para o terminal da direita não foram entregues. Assim que o terminal da direita foi iniciado, começa a receber os pacotes do terminal esquerdo (a partir do número 4). Da mesma forma, assim que o terminal direito foi ligado, o terminal esquerdo começa a receber seus pacotes (a partir do número 0). Adicionalmente, quando foi inserido o caractere “a” no terminal esquerdo, ele a seguir indica “Recebi do teclado: a”. O mesmo acontece quando, no terminal direito, insere-se o caractere “b”: ele logo em seguida imprime “Recebi do teclado: b”.

```
tividades/Repo-Github/CES-27/Atividade2/Tarefa3$ ./Processo 10003 10004
1 write udp 127.0.0.1:43710->127.0.0.1:10004: write: connection refused
Received 0 from 127.0.0.1:39949
3 write udp 127.0.0.1:43710->127.0.0.1:10004: write: connection refused
Received 1 from 127.0.0.1:39949
Received 2 from 127.0.0.1:39949
a
Recebi do teclado: a
Received 3 from 127.0.0.1:39949
Received 4 from 127.0.0.1:39949
Received 5 from 127.0.0.1:39949
Received 6 from 127.0.0.1:39949
Received 7 from 127.0.0.1:39949
^C
tividades/Repo-Github/CES-27/Atividade2/Tarefa3$ ./Processo 10004 10003
Received 4 from 127.0.0.1:43710
Received 5 from 127.0.0.1:43710
Received 6 from 127.0.0.1:43710
Received 7 from 127.0.0.1:43710
Received 8 from 127.0.0.1:43710
b
Recebi do teclado: b
Received 9 from 127.0.0.1:43710
Received 10 from 127.0.0.1:43710
9 write udp 127.0.0.1:39949->127.0.0.1:10003: write: connection refused
11 write udp 127.0.0.1:39949->127.0.0.1:10003: write: connection refused
^C
gustavo-nahum@debian-stretch:~/Documentos/ITA/Juliana/A
```

Figura 4. Teste de comunicação UDP entre dois programas: um com o servidor na porta 10004 (à esquerda) e o outro com o servidor na porta 10003 (à direita).

### Código Process3.go

```
package main

import (
    "fmt"
    "net"
    "os"
    "strconv"
    "time"
    "bufio"
)

var err string
var myPort string
var nServers int
var CliConn []*net.UDPConn

var ServConn *net.UDPConn

func CheckError(err error) {
    if err != nil {
        fmt.Println("Error: ", err)
    }
}

func readInput(ch chan string) {
    reader := bufio.NewReader(os.Stdin)
    for {
        text, _, _ := reader.ReadLine()
        ch <- string(text)
    }
}

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

    n,addr,err := ServConn.ReadFromUDP(buf)
    fmt.Println("Received ",string(buf[0:n]), " from ",addr)

    if err != nil {
        fmt.Println("Error: ",err)
    }
}

func doClientJob(otherProcess int, i int) {
```

```

msg := strconv.Itoa(i)
i++
buf := []byte(msg)
_,err := CliConn[otherProcess].Write(buf)
if err != nil {
    fmt.Println(msg, err)
}
time.Sleep(time.Second * 1)
}

func initConnections() {
    myPort = os.Args[1]
    nServers = len(os.Args) - 2

    CliConn = make([]*net.UDPConn, nServers)

    ServerAddr, err := net.ResolveUDPAddr("udp", ":" + myPort)
    CheckError(err)
    ServConn, err = net.ListenUDP("udp", ServerAddr)
    CheckError(err)

    LocalAddr, err := net.ResolveUDPAddr("udp", "127.0.0.1:0")
    CheckError(err)
    for i:=0; i < nServers; i++ {
        otherServerAddr,err := net.ResolveUDPAddr("udp","127.0.0.1:" + os.Args[i + 2])
        CheckError(err)

        CliConn[i], err = net.DialUDP("udp", LocalAddr, otherServerAddr)
        CheckError(err)
    }
}

func main() {
    initConnections()
    defer ServConn.Close()
    for i := 0; i < nServers; i++ {
        defer CliConn[i].Close()
    }

    i := 0
    ch := make(chan string)
    go readInput(ch)

    for {
        go doServerJob()
        select {
        case x, valid := <-ch:
            if valid {
                fmt.Printf("Recebi do teclado: %s \n", x)
            } else {
                fmt.Println("Channel closed!")
            }
        default:
            time.Sleep(time.Second * 1)
        }
        for j := 0; j < nServers; j++ {
            go doClientJob(j, i)
        }
        time.Sleep(time.Second * 1)
        i++
    }
}

```

## Tarefa 4

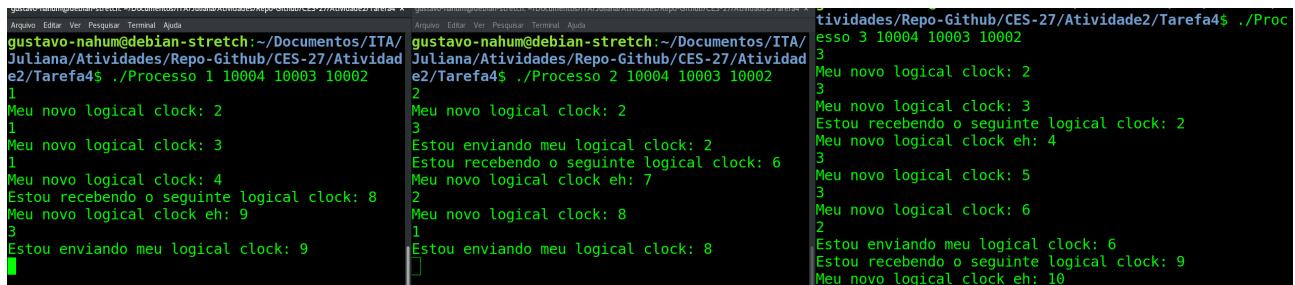
Nessa tarefa, foi necessário implementar uma simulação para os relógios lógicos escalares de Lamport. Dessa forma, em cada processo, havia uma variável inteira *logicalClock*, inicialmente com o valor 1, e que era incrementada a cada vez que era inserido no terminal o índice referente ao próprio processo. Alternativamente, poderia ser inserido o índice referente a algum dos outros processos: neste caso, o relógio lógico do processo corrente é enviado para o processo referenciado pelo índice inserido, e atualizado neste processo (passando a ser igual a  $1 + \max(\text{valor transmitido de relógio lógico}, \text{valor do relógio lógico do processo receptor})$ ).

Para isso, aproveitando-se a estrutura de canal e de leitura do buffer de IO implementada na tarefa 3, foi possível decodificar quais índices eram inseridos no terminal, e realizar as ações correspondentes em cada situação.

No caso em que o valor inserido correspondia ao índice do próprio processo, acrescia-se de um a variável inteira *logicalClock*, e imprimia-se o novo valor do relógio lógico.

Já no caso em que o valor inserido correspondia ao índice de algum dos outros processos, a função *doClientJob* envia o valor do *logicalClock* do processo corrente através da comunicação UDP com o processo referenciado pelo índice. A função *doServerJob* desse processo referenciado, por sua vez, recebia a mensagem e a armazenava na variável *receivedClock*. Comparando-se o *receivedClock* com o *logicalClock* do processo que recebeu a mensagem, determinava-se qual dentre estes números era o maior, e acrescia-se 1 a este número: este seria o novo valor de *logicalClock* do processo que recebeu a mensagem.

A execução desse programa em três terminais é ilustrada na figura 5.



```
gustavo-nahum@debian-stretch:~/Documentos/ITA/Juliana/Atividades/Repo-Github/CES-27/Atividade2/Tarefa4$ ./Processo 1 10004 10003 10002
1
Meu novo logical clock: 2
1
Meu novo logical clock: 3
1
Meu novo logical clock: 4
Estou recebendo o seguinte logical clock: 8
Meu novo logical clock eh: 9
3
Estou enviando meu logical clock: 9

gustavo-nahum@debian-stretch:~/Documentos/ITA/Juliana/Atividades/Repo-Github/CES-27/Atividade2/Tarefa4$ ./Processo 2 10004 10003 10002
2
Meu novo logical clock: 2
3
Estou enviando meu logical clock: 2
Estou recebendo o seguinte logical clock: 6
Meu novo logical clock eh: 7
2
Meu novo logical clock: 8
1
Estou enviando meu logical clock: 8

tividades/Repo-Github/CES-27/Atividade2/Tarefa4$ ./Processo 3 10004 10003 10002
3
Meu novo logical clock: 2
3
Meu novo logical clock: 3
Estou recebendo o seguinte logical clock: 2
Meu novo logical clock eh: 4
3
Meu novo logical clock: 5
3
Meu novo logical clock: 6
2
Estou enviando meu logical clock: 6
Estou recebendo o seguinte logical clock: 9
Meu novo logical clock eh: 10
```

Figura 5. Teste de comunicação UDP entre três programas: um com o servidor na porta 10004 (à esquerda), outro com o servidor na porta 10003 (ao centro) e o outro com o servidor na porta 10002 (à direita).

Verifica-se que, ao inserir no terminal 1 (à esquerda) o número 1, ele atualiza seu relógio lógico, que aumenta de 1 para 2. Analogamente, o mesmo acontece nos terminais 2 (ao centro) e 3 (à direita). Quando, no terminal 2, insere-se o número 3, ele envia o seu relógio lógico para o terminal 3; o terminal 3, por sua vez, imprime o valor recebido de relógio lógico, que é 2, compara com seu próprio relógio lógico, que é 3, determina o máximo entre estes valores, que é 3, e acresce 1, obtendo o novo valor de seu relógio lógico, que será 4. Acompanhando as operações listadas nos terminais, é possível verificar a sequência das evoluções dos relógios lógicos de cada processo.

Abaixo mostra-se o esquema do resultado da execução descrita.

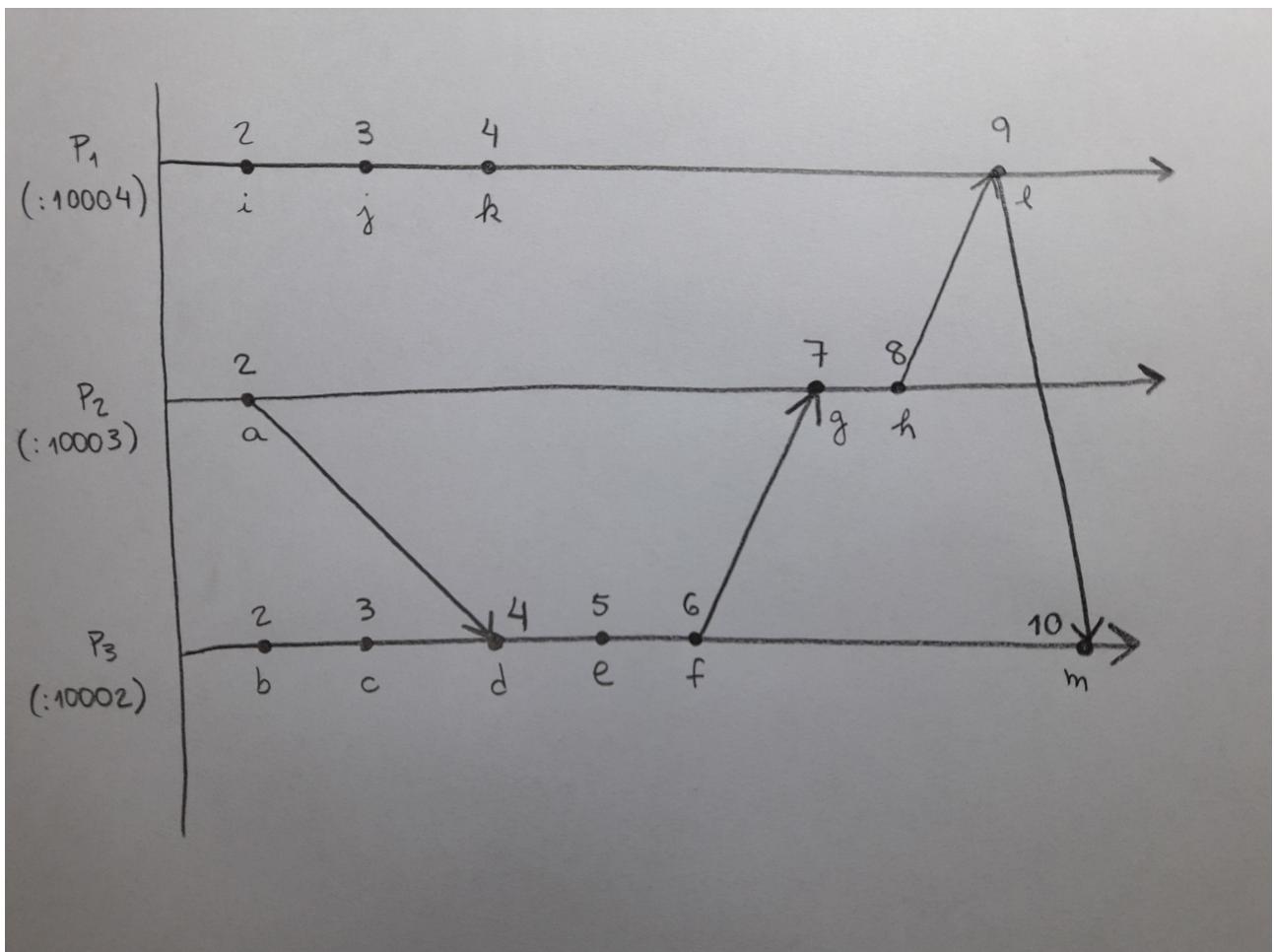


Figura 6. Ilustração da evolução dos relógios lógicos dos processos 1, 2 e 3 executados nos terminais mostrados na figura 5.

Código *Process4.go*:

```

package main

import (
    "fmt"
    "net"
    "os"
    "strconv"
    "time"
    "bufio"
)
var err error
var myProcess int
var myPort string
var allPorts []string
var nServers int
var CliConn []*net.UDPConn
var ServConn *net.UDPConn

var logicalClock int

func updateClock(clock1 int, clock2 int) int {
    if clock1 > clock2 {
        return clock1 + 1
    }
}

```

```

        return clock2 + 1
    }

func CheckError(err error) {
    if err != nil {
        fmt.Println("Error: ", err)
    }
}

func PrintError(err error) {
    if err != nil {
        fmt.Println("Error: ", err)
    }
}

func readInput(ch chan string) {
    reader := bufio.NewReader(os.Stdin)
    for {
        text, _, _ := reader.ReadLine()
        ch <- string(text)
    }
}

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

    n,_ ,err := ServConn.ReadFromUDP(buf)
    if err != nil {
        fmt.Println("Error: ",err)
    }
    fmt.Printf("Estou recebendo o seguinte logical clock: %s \n", string(buf[0:n]))
    receivedClock, err := strconv.Atoi(string(buf[0:n]))
    CheckError(err)
    logicalClock = updateClock(logicalClock, receivedClock)
    fmt.Printf("Meu novo logical clock eh: %d \n", logicalClock)
}

func doClientJob(otherProcess int, logClock int) {
    msg := strconv.Itoa(logClock)
    buf := []byte(msg)
    _,err := CliConn[otherProcess].Write(buf)
    if err != nil {
        fmt.Println(msg, err)
    }
    time.Sleep(time.Second * 1)
}

func initConnections() {
    myProcess, err = strconv.Atoi(os.Args[1])
    CheckError(err)
    myPort = os.Args[myProcess + 1]
    nServers = len(os.Args) - 2

    allPorts = make([]string, nServers + 1)
    for i:=1; i <= nServers; i++ {
        allPorts[i] = os.Args[i + 1]
    }

    CliConn = make([]*net.UDPConn, nServers + 1)

    ServerAddr, err := net.ResolveUDPAddr("udp", ":" + myPort)
    CheckError(err)
    ServConn, err = net.ListenUDP("udp", ServerAddr)
}

```

```

CheckError(err)

LocalAddr, err := net.ResolveUDPAddr("udp", "127.0.0.1:0")
CheckError(err)
for i:=1; i <= nServers; i++ {
    if i != myProcess {
        otherServerAddr,err := net.ResolveUDPAddr("udp","127.0.0.1:" + allPorts[i])
        CheckError(err)

        CliConn[i], err = net.DialUDP("udp", LocalAddr, otherServerAddr)
        CheckError(err)
    }
}
}

func main() {
    initConnections()
    defer ServConn.Close()
    for i := 1; i <= nServers; i++ {
        if i != myProcess {
            defer CliConn[i].Close()
        }
    }

    i := 0
    ch := make(chan string)
    go readInput(ch)
    logicalClock = 1

    for {
        go doServerJob()
        select {
        case x, valid := <-ch:
            if valid {
                if x == strconv.Itoa(myProcess) {
                    logicalClock++
                    fmt.Printf("Meu novo logical clock: %d \n", logicalClock)
                } else if _, err := strconv.Atoi(x); err == nil {
                    fmt.Printf("Estou enviando meu logical clock: %d \n", logicalClock)
                    x, err := strconv.Atoi(x)
                    CheckError(err)
                    go doClientJob(x, logicalClock)
                }
            } else {
                fmt.Println("Channel closed!")
            }
        default:
            time.Sleep(time.Second * 1)
        }
        time.Sleep(time.Second * 1)
        i++
    }
}

```

## Tarefa 5

Nesta última tarefa, implementaram-se relógios lógicos de Lamport vetoriais, isto é, agora os relógios lógicos armazenaram as informações referentes aos seus próprios processos, bem como dos demais com os quais se comunicam. Para isso, foi necessário criar em Go uma struct que armazenasse tanto o índice que referenciava o seu processo, como um slice de inteiros que registrasse os relógios vetoriais de todos os processos.

Novamente, quando digitava-se em um terminal o número correspondente ao índice do processo executado nesse terminal, esse processo incrementava o seu componente do relógio lógico em 1.

Já quando se digitava em um terminal o número correspondente ao índice de um processo executado em outro terminal, o terminal que recebeu a entrada digitada transmitia, via comunicação UDP, o seu relógio vetorial completo de forma serializada, usando json. Para isso, foi necessário no código de *doClientJob* incluir a função *json.Marshal()*, que recebia como argumento a struct contendo o relógio vetorial a ser serializado, e produzia como saída a mensagem de requisição json, que por sua vez seria transmitida pela comunicação UDP. Para que essa mensagem json fosse decodificada no servidor, este fazia uso da função *json.Unmarshal*, que depositava na struct *receivedVectorClock* o conteúdo da struct que foi transmitida. Comparando-se os valores que as structs possuíam em cada posição, era possível tomar o maior deles (e, no caso da componente do relógio lógico correspondente ao índice do processo que recebeu a mensagem, também aumentava-se 1), e atribuía-se esse novo valor ao relógio vetorial atualizado do processo que recebeu a mensagem.

A execução desse programa em três terminais é ilustrada na figura 7.

```

Ativado Editar Ver Pausar Terminal Ajuda
gustavo-nahum@debian-stretch:~/Documentos/ITA/Juliana/Atividades/Repo-Github/CES-27/Atividade2/Tarefa5$ ./Process5 1 10004 10003 10002
1
Meu novo vector clock: (2, 0, 0)
1
Meu novo vector clock: (3, 0, 0)
1
Meu novo vector clock: (4, 0, 0)
3
Estou enviando meu vector clock: (4, 0, 0)
2
Relogio vetorial recebido: (4, 5, 7)
Meu relogio vetorial atualizado: (4, 4, 7)
2
Meu novo vector clock: (4, 5, 7)
1
Estou enviando meu vector clock: (4, 5, 7)
3
Relogio vetorial recebido: (4, 5, 7)
Meu relogio vetorial atualizado: (4, 5, 7)
2
Meu novo vector clock: (4, 5, 7)

tividades/Repo-Github/CES-27/Atividade2/Tarefa5$ ./Process5 2 10004 10003 10002
2
Meu novo vector clock: (0, 2, 0)
2
Meu novo vector clock: (0, 3, 0)
3
Relogio vetorial recebido: (4, 0, 7)
Meu relogio vetorial atualizado: (4, 4, 7)
2
Meu novo vector clock: (0, 4, 0)
3
Meu novo vector clock: (0, 5, 0)
2
Meu novo vector clock: (4, 0, 6)
1
Relogio vetorial recebido: (4, 0, 6)
Meu relogio vetorial atualizado: (4, 0, 7)
3
Meu novo vector clock: (4, 0, 7)

tividades/Repo-Github/CES-27/Atividade2/Tarefa5$ ./Process5 3 10004 10003 10002
3
Meu novo vector clock: (0, 0, 2)
3
Meu novo vector clock: (0, 0, 3)
3
Meu novo vector clock: (0, 0, 4)
3
Meu novo vector clock: (0, 0, 5)
2
Estou enviando meu vector clock: (4, 0, 7)
1
Relogio vetorial recebido: (4, 0, 7)
Meu relogio vetorial atualizado: (4, 0, 8)
3
Meu novo vector clock: (4, 0, 8)
2
Estou enviando meu vector clock: (4, 0, 8)

```

Figura 7. Teste de comunicação UDP entre três programas: um com o servidor na porta 10004 (à esquerda), outro com o servidor na porta 10003 (ao centro) e o outro com o servidor na porta 10002 (à direita).

Verifica-se que, ao digitar o número do próprio terminal, a componente do relógio vetorial referente ao próprio processo é acrescida de 1. Quando o terminal 1 recebe a entrada “3”, envia o seu relógio com valores (4, 0, 0), enquanto que no terminal 3 o relógio estava em (0, 0, 5); assim, o relógio atualizado torna-se (4, 0, 6). A seguir, o terminal 3 envia seu relógio para 2, quando seu estado era (4, 0, 7); o relógio de 2 estava nesse momento em (0, 0, 3), portanto torna-se (4, 5, 7). As demais transições seguem esse mesmo padrão, que está de acordo com o comportamento desejado dos relógios lógicos vetoriais.

Abaixo mostra-se o esquema do resultado da execução descrita.

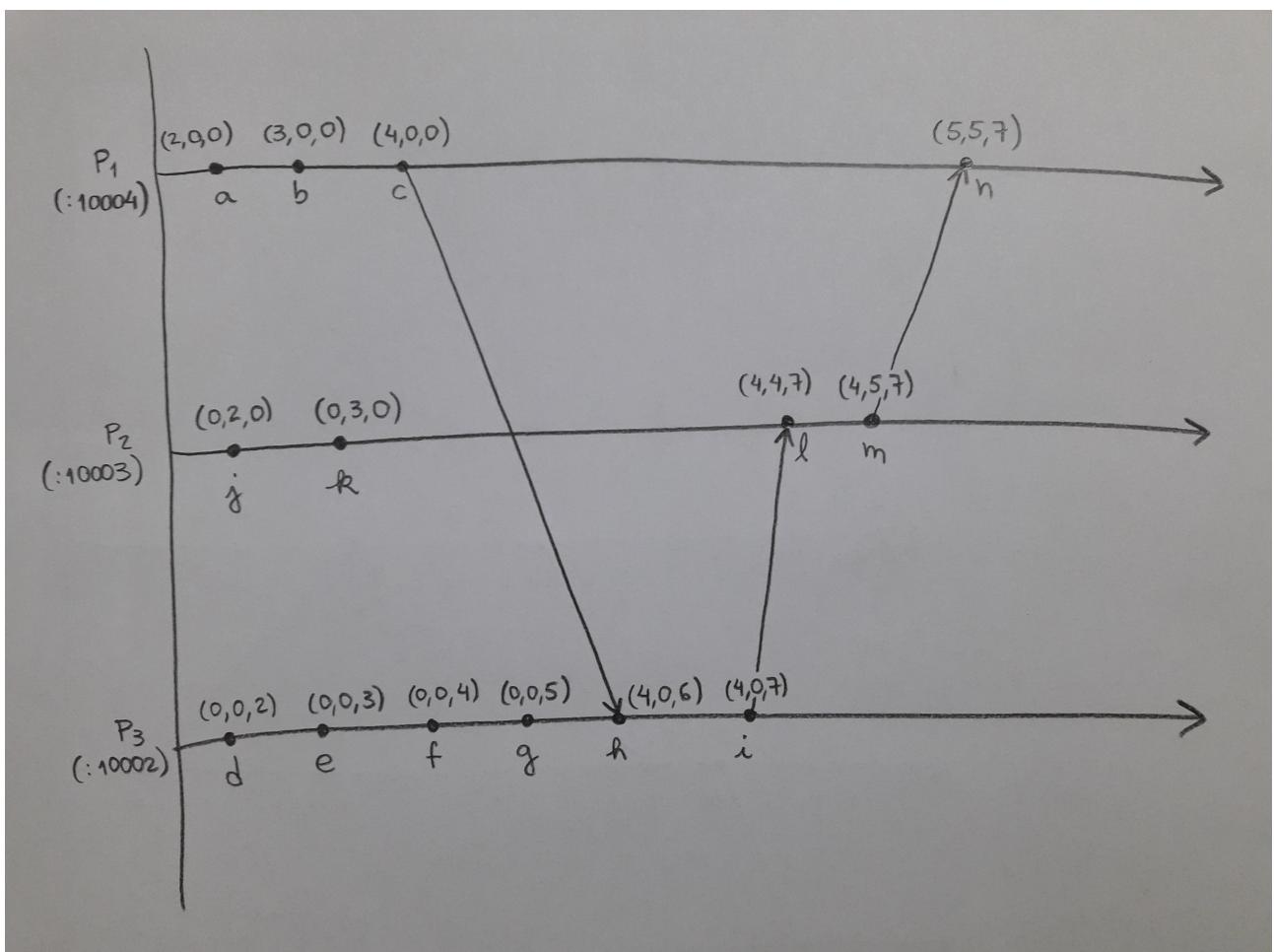


Figura 8. Ilustração da evolução dos relógios lógicos dos processos 1, 2 e 3 executados nos terminais mostrados na figura 7.

Código *Process5.go*:

```
package main

import (
    "fmt"
    "net"
    "os"
    "strconv"
    "time"
    "bufio"
)

var err error
var myProcess int
var myPort string
var allPorts [] string
var nServers int
var CliConn []*net.UDPConn
var ServConn *net.UDPConn

var logicalClock int

func updateClock(clock1 int, clock2 int) int {
    if clock1 > clock2 {
        return clock1 + 1
    }
    return clock2 + 1
}

func CheckError(err error) {
    if err != nil {
        fmt.Println("Error: ", err)
    }
}

func PrintError(err error) {
    if err != nil {
        fmt.Println("Error: ", err)
    }
}

func readInput(ch chan string) {
    reader := bufio.NewReader(os.Stdin)
    for {
        text, _, _ := reader.ReadLine()
        ch <- string(text)
    }
}

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

    n, _, err := ServConn.ReadFromUDP(buf)
    if err != nil {
        fmt.Println("Error: ", err)
    }
    fmt.Printf("Estou recebendo o seguinte logical clock: %s \n", string(buf[0:n]))
    receivedClock, err := strconv.Atoi(string(buf[0:n]))
    CheckError(err)
    logicalClock = updateClock(logicalClock, receivedClock)
    fmt.Printf("Meu novo logical clock eh: %d \n", logicalClock)
```

```

}

func doClientJob(otherProcess int, logClock int) {
    msg := strconv.Itoa(logClock)
    buf := []byte(msg)
    _,err := CliConn[otherProcess].Write(buf)
    if err != nil {
        fmt.Println(msg, err)
    }
    time.Sleep(time.Second * 1)
}

func initConnections() {
    myProcess, err = strconv.Atoi(os.Args[1])
    CheckError(err)
    myPort = os.Args[myProcess + 1]
    nServers = len(os.Args) - 2

    allPorts = make([]string, nServers + 1)
    for i:=1; i <= nServers; i++ {
        allPorts[i] = os.Args[i + 1]
    }

    CliConn = make([]*net.UDPConn, nServers + 1)

    ServerAddr, err := net.ResolveUDPAddr("udp", ":" + myPort)
    CheckError(err)
    ServConn, err = net.ListenUDP("udp", ServerAddr)
    CheckError(err)

    LocalAddr, err := net.ResolveUDPAddr("udp", "127.0.0.1:0")
    CheckError(err)
    for i:=1; i <= nServers; i++ {
        if i != myProcess {
            otherServerAddr,err := net.ResolveUDPAddr("udp","127.0.0.1:" + allPorts[i])
            CheckError(err)

            CliConn[i], err = net.DialUDP("udp", LocalAddr, otherServerAddr)
            CheckError(err)
        }
    }
}

func main() {
    initConnections()
    defer ServConn.Close()
    for i := 1; i <= nServers; i++ {
        if i != myProcess {
            defer CliConn[i].Close()
        }
    }

    i := 0
    ch := make(chan string)
    go readInput(ch)
    logicalClock = 1

    for {
        go doServerJob()
        select {
        case x, valid := <-ch:
            if valid {
                if x == strconv.Itoa(myProcess) {

```

```
    logicalClock++
    fmt.Printf("Meu novo logical clock: %d \n", logicalClock)
} else if _, err := strconv.Atoi(x); err == nil {
    fmt.Printf("Estou enviando meu logical clock: %d \n", logicalClock)
    x, err := strconv.Atoi(x)
    CheckError(err)
    go doClientJob(x, logicalClock)
}
} else {
    fmt.Println("Channel closed!")
}
default:
    time.Sleep(time.Second * 1)
}
time.Sleep(time.Second * 1)
i++
}
}
```