

1º ATIVIDADE de CES-27 / 2018

CTA - ITA - IEC

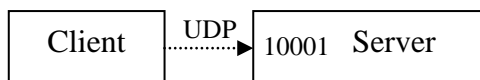
Prof Hirata e Prof Juliana

Objetivo: Simular processos rodando e trocando seus relógios lógicos (Logical Clock definido por Lamport).

Entregar (através do GoogleClassroom): Códigos dos exercícios (arquivos .go) e relatório. Favor NÃO compactar os arquivos. O relatório deve explicar particulares do código das **TAREFAS** solicitadas, apresentar testes realizados (com *print* de telas) e comentar os resultados encontrados (comparando com os resultados esperados).

Dica 1: Compreenda o funcionamento do programa cliente-servidor usando conexão UDP, como descrito no link abaixo.

<https://varshneyabhi.wordpress.com/2014/12/23/simple-udp-clientserver-in-golang/>



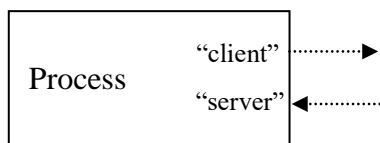
A ideia aqui é o `Server` enviar indefinidamente valores inteiros (em ordem crescente) para o `Client`.

Obs: Veja que no código a porta 10001 é fixa. É a porta que o servidor “escuta”.

Teste o sistema assim:

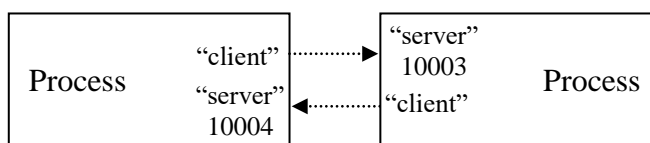
- Terminal 1: `Client`
 - Terminal 2: `Server`
-

Dica 2: Junte num só arquivo `Process.go` o código de cliente e servidor. Assim o processo terá capacidade de enviar mensagens (pelo cliente) e receber mensagens (pelo servidor).



A ideia é termos vários processos se comunicando. Então precisamos receber como parâmetro (no momento de executar o programa) a porta do servidor do processo em questão (primeiro valor na chamada abaixo) e as portas dos servidores dos demais processos (demais valores na chamada abaixo). Exemplo com dois processos:

- Terminal 1: `Process :10004 :10003`
- Terminal 2: `Process :10003 :10004`



A ideia aqui é um `Process` enviar indefinidamente valores inteiros (em ordem crescente) para todos os demais `Process`.

Obs: O endereço 127.0.0.1 pode continuar fixo no código. Caso queira testar em diferentes máquinas, isso deve ser configurável.

Uma dica para organizar seu código:

```
package main

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

//Variáveis globais interessantes para o processo
var err string
var myPort string //porta do meu servidor
var nServers int  //qtde de outros processo
var CliConn []*net.UDPConn //vetor com conexões para os servidores
                                //dos outros processos
var ServConn *net.UDPConn  //conexão do meu servidor (onde recebo
                                //mensagens dos outros processos)

func CheckError(err error) {
    if err != nil {
        fmt.Println("Erro: ", err)
        os.Exit(0)
    }
}

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

func doServerJob() {
    //Ler (uma vez somente) da conexão UDP a mensagem
    //Escrever na tela a msg recebida (indicando o endereço de quem
    //enviou)
}

func doClientJob(otherProcess int, i int) {
    //Enviar uma mensagem (com valor i) para o servidor do processo
    //otherServer
}

func initConnections() {
    myPort = os.Args[1]
    nServers = len(os.Args) - 2
    /*Esse 2 tira o nome (no caso Process) e tira a primeira porta
    (que é a minha). As demais portas são dos outros processos*/

    //Outros códigos para deixar ok a conexão do meu servidor (onde
    //recebo msgs). O processo já deve ficar habilitado a receber
    //msgs.

    //Outros códigos para deixar ok as conexões com os servidores
    //dos outros processos. Colocar tais conexões no vetor CliConn.
}
```

```

func main() {

    initConnections()

    //O fechamento de conexões deve ficar aqui, assim só fecha
    //conexão quando a main morrer
    defer ServConn.Close()
    for i := 0; i < nServers; i++ {
        defer CliConn[i].Close()
    }

    //Todo Process fará a mesma coisa: ouvir msg e mandar infinitos
    //i's para os outros processos
    i := 0
    for {
        //Server
        go doServerJob()
        //Client
        for j := 0; j < nServers; j++ {
            go doClientJob(j, i)
        }
        // Wait a while
        time.Sleep(time.Second * 1)
        i++
    }
}

```

Teste o sistema assim:

- Terminal 1: Process :10004 :10003
- Terminal 2: Process :10003 :10004

Teste o sistema assim:

- Terminal 1: Process :10002 :10003 :10004
- Terminal 2: Process :10003 :10002 :10004
- Terminal 3: Process :10004 :10002 :10003

Da forma implementada, a primeira porta é a do processo corrente e as demais portas (em qualquer ordem) são dos outros processos.

Dica 3: Queremos “controlar” cada processo de modo independente para ele fazer uma ação que desejarmos. Assim eles não terão o comportamento padrão de antes e poderemos simular diferentes situações. Vamos “controlar” o processo através do envio de texto pela janela de comando (do terminal em que o processo roda).

Para isso, adicione a função abaixo ao código do processo. Lembre-se de importar a biblioteca “bufio”.

```
func readInput(ch chan string) {  
    // Non-blocking async routine to listen for terminal input  
    reader := bufio.NewReader(os.Stdin)  
    for {  
        text, _, _ := reader.ReadLine()  
        ch <- string(text)  
    }  
}
```

Substitua o comportamento anterior do processo (parte da ‘main’) pelo seguinte:

```
for {  
    //Server  
    go doServerJob()  
  
    // When there is a request (from stdin). Do it!  
    select {  
    case x, valid := <-ch:  
        if valid {  
            fmt.Printf("Recebi do teclado: %s \n", x)  
            //Client  
            for j := 0; j < nServers; j++ {  
                go doClientJob(j, 100)  
            }  
        } else {  
            fmt.Println("Channel closed!")  
        }  
    default:  
        // Do nothing in the non-blocking approach.  
        time.Sleep(time.Second * 1)  
    }  
}
```

Note que agora o processo pode receber mensagens dos outros processos, mas a ação dele é de acordo com o recebido pelo terminal. Se o processo receber algo (por exemplo, um caractere) pelo teclado, ele envia o valor 100 para todos os demais processos.

Teste o sistema assim:

- Terminal 1: Process :10004 :10003
 - Terminal 2: Process :10003 :10004
 - Terminal 1: a
 - Terminal 2: b
-

Abaixo seguem as tarefas propostas.!

Tarefa 1: Relógio Lógico Escalar

Finalmente vamos implementar uma simulação para o Relógio Lógico Escalar de Lamport. Vamos seguir o algoritmo indicado nos slides da aula (slides 12 e 13). Para isso:

- Considere que vamos rodar a simulação assim (obs: no caso de dois processos):

Terminal 1: Process 1 :10004 :10003

Terminal 2: Process 2 :10004 :10003

Nesse caso, temos sempre a mesma sequência de portas. Cada processo tem seu *id*. De acordo com o *id*, o processo sabe sua porta e a dos colegas. Nesse exemplo o processo 1 usa tem porta 10004 e o processo 2 tem porta 10003.

Considere que *id* começa em 1. Os ids são sempre valores consecutivos (adicionando 1).

- Cada processo terá seu *logicalClock* que inicia em 0.
- No terminal, você pode solicitar que o processo envie mensagem para outro. Ex:

Terminal 1: 2

- ✓ Nesse caso, o processo 1 envia uma mensagem ao processo 2. A mensagem é apenas o seu *logicalClock* (devidamente atualizado).
- ✓ Processo 1 deve imprimir o valor enviado no seu terminal.
- ✓ Quando o processo 2 receber a mensagem, ele deve imprimir o valor recebido e imprimir também seu *logicalClock* (devidamente atualizado).

- No terminal, você pode solicitar que o processo execute uma ação interna. Ex:

Terminal 1: 1

- ✓ Nesse caso, o valor recebido é o próprio *id* do processo. O processo apenas incrementa o seu *logicalClock* e imprime esse novo valor.

Para o relatório ⇒ Teste o sistema assim com três processos:

- Terminal 1: Process 1 :10004 :10003 :10002
 - Terminal 2: Process 2 :10004 :10003 :10002
 - Terminal 3: Process 3 :10004 :10003 :10002
 - Trabalhe com o caso apresentado na aula (slide 13) para mostrar a evolução dos relógios. Mostre o esquema (figura) do resultado esperado e apresente o resultado obtido (telas). Explique/comente.
 - Elabore um novo caso e comente! Pode usar mais processos!
-

Tarefa 2: Relógio Lógico Vetorial

Vamos manter a ideia anterior, mas agora cada processo irá guardar o relógio lógico de todos os processos. Considere o algoritmo descrito na aula (slide 24).

Defina uma estrutura de dados (struct) com *id* do processo corrente e um vetor de relógios de todos os processos. Ex:

```
type ClockStruct struct {  
    id      int  
    clocks []int  
}  
var logicalClock ClockStruct
```

Para enviar a struct via UDP, você deve trabalhar com serialização. Go tem suporte (através de bibliotecas standard) para json e gob.

Referências:

<http://www.ugorji.net/blog/serialization-in-go>

<https://gist.github.com/reterVision/33a72d70194d4a3c272e>

Para o relatório ⇒ Teste o sistema assim com três processos:

- Terminal 1: Process 1 :10004 :10003 :10002
- Terminal 2: Process 2 :10004 :10003 :10002
- Terminal 3: Process 3 :10004 :10003 :10002
- Trabalhe com o caso apresentado em aula (slide 24) para mostrar a evolução dos relógios. Mostre o esquema (figura) do resultado esperado e apresente o resultado obtido (telas). Explique/comente.
- Elabore um novo caso e comente! Pode usar mais processos!

Bom trabalho!