ATIVIDADE DIRIGIDA de CES-27

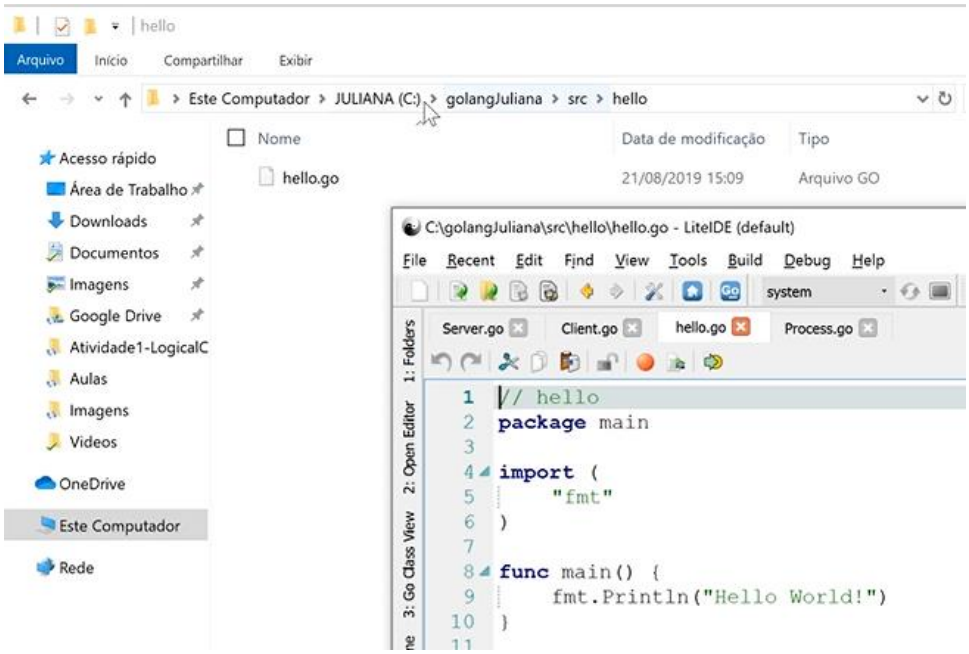
CTA - ITA - IEC

Prof Hirata e Prof Juliana

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

Início: Escolha uma IDE para programar em Go e instale o Go localmente. Rode o hello.go para ver se funciona.

- https://golang.org/doc/install
- https://golang.org/doc/tutorial/getting-started

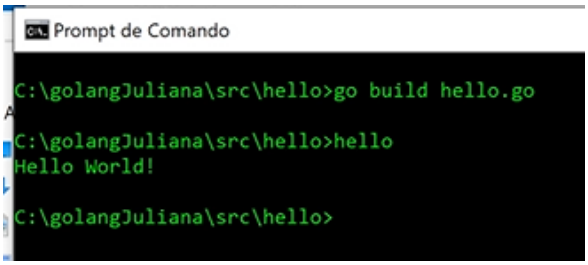


No caso acima, temos:

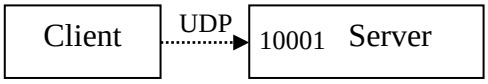
- LiteIDE para escrever o código, mas vocês podem escolher a IDE que desejarem.
- Variável de ambiente `GOPATH` (que representa o *workspace*) como "C:\golangJuliana"
  - Após definir essa variável, pode testar no terminal: `cd %GOPATH%`
- Os códigos devem ficar em `GOPATH\src`
  - Aqui `hello.go` está em `GOPATH\src\hello`

Para funcionar:

- `go run hello.go` : compila e roda
- Na figura abaixo fizemos build (para gerar o `exe`) e depois chamamos o `exe`. Vamos fazer assim, em geral, pois trabalharemos com diferentes terminais.



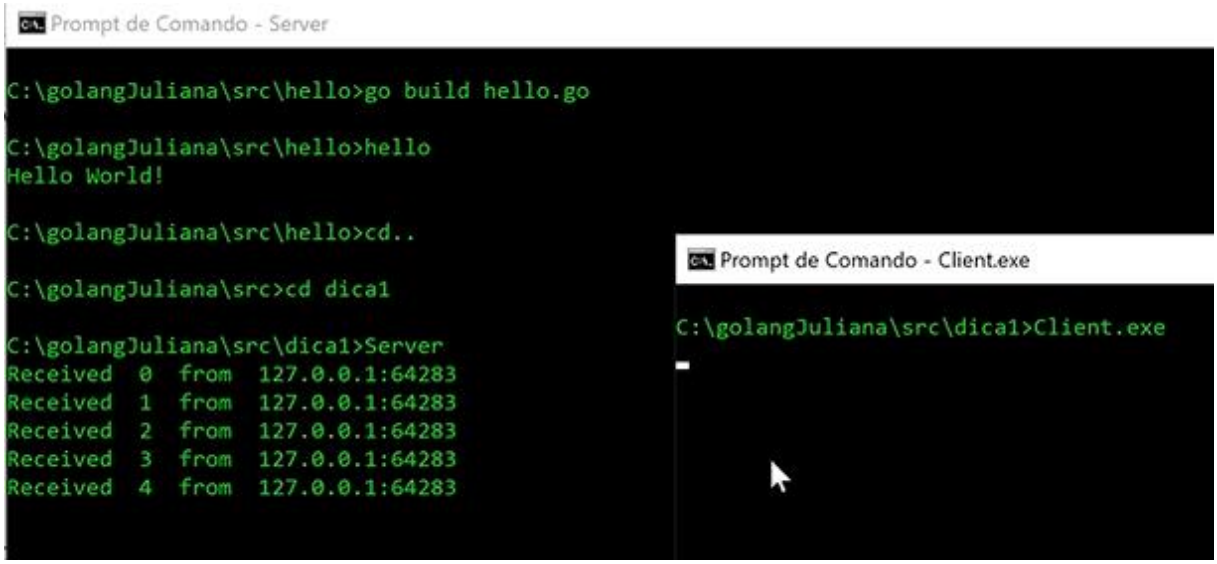
Dica 1: Compreenda o funcionamento do programa cliente-servidor usando conexão UDP, como descrito no link abaixo.
https://varshneyabhi.wordpress.com/2014/12/23/simple-udp-clientserver-in-golang/



A ideia aqui é o `Client` enviar indefinidamente valores inteiros (em ordem crescente) para o `Server`.
Obs: Veja que no código a porta 10001 é fixa. É a porta que o servidor "escuta".

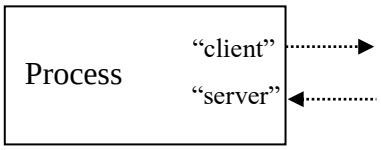
Teste o sistema assim:

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



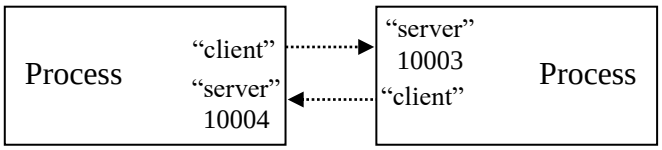
Resultado esperado (de acordo com o código atual): `Client` não imprime nada e `Server` imprime os números recebidos.

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).



Desejamos ter 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.

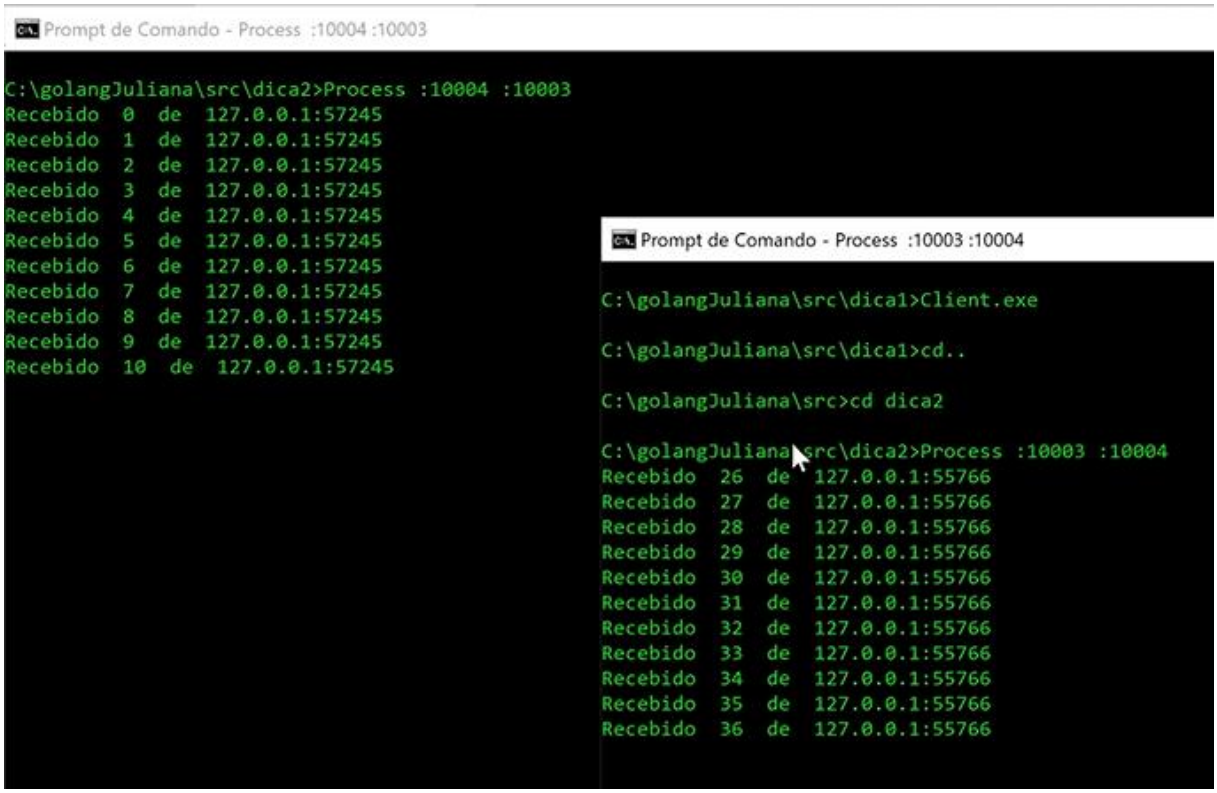
Abaixo consta uma sugestão para o código `Process.go`. Basta completar as funções `doServerJob e doClientJob`.

```go
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() {
      //Loop infinito mesmo
      for {
            //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*/
        CliConn = make([]*net.UDPConn, nServers)

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

        ServerAddr, err := net.ResolveUDPAddr("udp", "127.0.0.1"+myPort)
        CheckError(err)
        ServConn, err = net.ListenUDP("udp", ServerAddr)
        CheckError(err)

       //Outros códigos para deixar ok as conexões com os servidores dos
       outros processos. Colocar tais conexões no vetor CliConn.
        for servidores := 0; servidores < nServers; servidores++ {
                ServerAddr, err := net.ResolveUDPAddr("udp",
                "127.0.0.1"+os.Args[2+servidores])
                CheckError(err)

                Conn, err := net.DialUDP("udp", nil, ServerAddr)
                CliConn[servidores] = Conn
                CheckError(err)
        }
}

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: ficar ouvindo mensagens e man-
        dar infinitos i's para os outros processos
        go doServerJob()
        i := 0
        for {
                for j := 0; j < nServers; j++ {
                        go doClientJob(j, i)
                }
                // Espera um pouco
                time.Sleep(time.Second * 1)
                i++
        }
}
```

Teste o sistema assim:

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



Resultado esperado: Perceba que o segundo processo (iniciado depois) não imprime os valores iniciais (no caso, após 26). Isso porque ele começou depois e perdeu o que o outro processo o enviou.

Depois 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) {
        // Rotina não-bloqueante que "escuta" o stdin
        reader := bufio.NewReader(os.Stdin)
        for {
                text, _, _ := reader.ReadLine()
                ch <- string(text)
        }
}
```

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

```
    ch := make(chan string) //canal que guarda itens lidos do teclado
    go readInput(ch) //chamar rotina que "escuta" o teclado

    go doServerJob()
    for {
            // Verificar (de forma não bloqueante) se tem algo no
            // stdin (input do terminal)
            select {
            case x, valid := <-ch:
                    if valid {
                        fmt.Printf("Recebi do teclado: %s \n", x)
                        for j := 0; j < nServers; j++ {
                             go doClientJob(j, 100)
                        }
                    } else {
                        fmt.Println("Canal fechado!")
                    }
            default:
                    // Fazer nada...
                    // Mas não fica bloqueado esperando o teclado
                    time.Sleep(time.Second * 1)
            }

            // Esperar um pouco
            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

Processo 1 começa a enviar 100 aos colegas (no caso, somente o Processo 2)

Processo 2 começa a enviar 100 aos colegas (no caso, somente o Processo 1)

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 sequencia 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.

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 ver a evolução dos relógios.

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
```

Obs: Variáveis *Id* e *Clocks* devem ficar com letras maiúsculas para evitar problema com json. Dica da turma anterior!

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

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 ver a evolução dos relógios.

Bom estudo!