2º ATIVIDADE de CES-27 / 2018

CTA - ITA - IEC

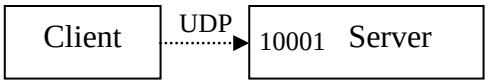
Prof Juliana e Prof Vitor

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

Entregar (através do TIDIA): Códigos dos exercícios (arquivos .go) e relatório. O relatório deve apresentar o código, explicar detalhes particulares/críticos do código, apresentar testes realizados e comentar os resultados.

Tarefa 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/



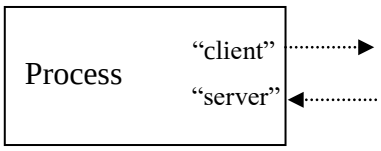
Obs: Veja que no código a porta 10001 é fixa. É a porta que o servidor "escuta".

Para o relatório ⇨ Teste o sistema assim:

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

Tarefa 2:

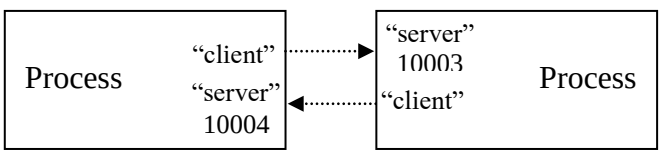
Junte num só arquivo `Process.go` o código de cliente e servidor. Assim o processo terá capacidade de enviar mensagens (pelo cliente) e receber mensagens (pelo servidor).



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 e as portas dos servidores dos demais processos. Exemplo com dois processos:

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

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
      //Escreve na tela a msg recebida
}
func doClientJob(otherProcess int, i int) {
      //Envia 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
      //Outros códigos para deixar ok as conexões com os servidores
      dos outros processos
}
```

```
func main() {

        initConnections()
        //O fechamento de conexões devem 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 infini-
        tos 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++
        }
}
```

Para o relatório ⇨Teste o sistema assim:

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

Para o relatório ⇨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.

Tarefa 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)
                        } else {
                           fmt.Println("Channel closed!")
                        }
                default:
                        // Do nothing in the non-blocking approach.
                        time.Sleep(time.Second * 1)
                }
                // Wait a while
                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.

Para o relatório ⇨Teste o sistema assim:

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

Tarefa 4: Relógio Lógico Escalar

Finalmente vamos implementar uma simulação para o relógio lógico escalar de Lamport. 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.
- Cada processo terá seu *logicalClock* que inicia em 1.
- 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*.
  - ✓ Processo 1 deve imprimir o valor enviado no seu terminal.
  - ✓ Quando o processo 2 receber a mensagem, ele deve atualizar seu *logicalClock* com 1 + Maior entre o clock da mensagem e próprio clock. Ele deve imprimir o valor recebido e o seu novo clock.
- 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 1 no 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`
- Elabore um caso para mostrar a evolução dos relógios. Mostre o esquema (figura) do resultado esperado e apresente o resultado obtido (telas). Explique/comente.

Tarefa 5: Relógio Lógico Vetorial

Vamos manter a ideia anterior, mas agora cada processo irá guardar o relógio lógico de todos os processos.

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

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`
- Elabore um caso para mostrar a evolução dos relógios. Mostre o esquema (figura) do resultado esperado e apresente o resultado obtido (telas). Explique/comente.

Bom trabalho!