

1º LAB de CES-27 / CE-288

CTA - ITA - IEC

Prof Hirata e Prof Juliana

Objetivo: Trabalhar com algoritmo de exclusão mútua para sistemas distribuídos.

Entregar (através do Google Classroom): Códigos dos exercícios (arquivos .go) e relatório. Favor NÃO compactar os arquivos. O relatório deve explicar particulares do código da tarefa solicitada, apresentar casos de testes realizados e comentar os resultados encontrados (comparando com os resultados esperados). Caprichem!

Tarefa:

Implemente o Algoritmo de Ricart-Agrawala para exclusão mútua.
Atenção: Esse algoritmo usa relógio lógico ESCALAR.

Obs: CS = *critical section*

Requisitos:

- 1) Vamos rodar o sistema como abaixo descrito. Neste exemplo, temos 3 processos (`Process.go`) e um recurso compartilhado (`SharedResource.go`):
 - Terminal A: `SharedResource`
 - Terminal B: `Process 1 :10002 :10003 :10004`
 - Terminal C: `Process 2 :10002 :10003 :10004`
 - Terminal D: `Process 3 :10002 :10003 :10004`

Obs: 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. Ex: o processo 1 “escuta” na porta 10002, o processo 2 “escuta” na porta 10003, e o processo 4 “escuta” na porta 10004.

Obs: O algoritmo de Ricart-Agrawala é implementado no processo (`Process.go`).

- 2) O recurso compartilhado será representado por um processo também, chamado `SharedResource.go`. Basicamente o `SharedResource` fica esperando alguém mandar uma mensagem para ele, através de uma porta fixa (ex: 10001). A mensagem recebida contém: o processo (*id*) que a enviou, o relógio lógico desse processo (o *sender*), e um texto simples (ex: “Oi”). O código do `SharedResource` é simples como indicado abaixo.

```
func main() {
    Address, err := net.ResolveUDPAddr("udp", ":10001")
    CheckError(err)
    Connection, err := net.ListenUDP("udp", Address)
    CheckError(err)
    defer Connection.Close()
    for {
        //Loop infinito para receber mensagem e escrever todo
        //conteúdo (processo que enviou, relógio recebido e texto)
        //na tela
        //FALTA FAZER
    }
}
```

3) O processo (`Process.go`) deve “escutar” o teclado (terminal).

3.1) Caso receba a mensagem “x”, o processo deve solicitar acesso à CS, em seguida usar a CS e liberar a CS.

- Caso o processo receba “x” indevido, ele deve imprimir na tela “x ignorado”. O “x” é indevido quando o processo já está na CS ou esperando para obter a CS.
- O controle para acessar a CS deve justamente ser feito com o Algoritmo de Ricart-Agrawala!

3.2) Caso receba o seu *id*, o processo executa uma ação interna (apenas um incremento do seu relógio lógico).

- Isso serve apenas para testarmos o algoritmo depois com relógios distintos para os processos.

4) “Usar a CS” (em `Process.go`) significa simplesmente enviar uma mensagem para o `SharedResource.go` e dormir um pouco.

- Assim que conseguir o acesso a CS, o processo deve escrever na sua tela “Entre na CS”.
- Antes de liberar a CS, o processo deve escrever na sua tela “Sai da CS”.

5) O processo deve sempre ser capaz de receber mensagens dos outros processos.

- Assim o processo pode estar esperando a CS, mas receber a mensagem de outro processo solicitando a CS.
 - Provavelmente você terá que usar `goroutine` para a função `doServerJob`. E ainda essa função terá um loop infinito para receber mensagens dos outros processos.

Convenção sobre como atualizar o clock:

O algoritmo de Ricart-Agrawala usa relógio lógico escalar, mas não indica onde/como atualizá-lo. Existem diferentes formas de atualizar o clock. Abaixo segue uma proposta. A ideia é não aumentar o clock sem necessidade, e garantir que o processo esteja atualizado em relação aos demais.

- a) Quando houver um ação interna, incrementar o clock.
- b) Quando for enviar os *requests*, incrementar o clock e mandar esse valor para todos os demais processos. Obs: Todos vão receber o mesmo valor!
- c) Quando receber um *request*, atualizar o clock para ficar coerente com quem enviou: $I + \max(my\ clock, received\ clock)$.
- d) Quando enviar um *reply*, não incrementar o clock. Obs: Mande o seu clock atual mesmo.
- e) Quando receber um *reply*, atualizar o clock para ficar coerente com quem enviou: $I + \max(my\ clock, received\ clock)$.

Atenção: Provavelmente você terá diferentes goroutines alterando o clock, logo convém utilizar um **mutex** para garantir a correta manipulação desta variável. Referências sobre mutex: <https://golangbot.com/mutex/> e <https://tour.golang.org/concurrency/9>

Casos de teste (para o relatório):

- **Caso 1:** Elabore um caso trivial com um processo solicitando a CS e, depois que ele liberar, outro processo solicita a CS.
- **Caso 2:** Elabore um caso com processos solicitando a CS “simultaneamente”. A ideia é mostrar que somente um processo acessa por vez.
- **Caso 3:** Elabore um caso que caia especificamente na condição “ $state=WANTED$ and $(T, p_j) < (T_i, p_i)$ ”. A ideia é mostrar que um processo que pediu a CS antes, terá a preferência.

Para cada caso:

- Use três ou mais processos.
- Mostre o esquema (figura) do resultado esperado. A figura 1 é um bom exemplo disso.
- Apresente o resultado obtido (*print* de suas telas).
- Explique/comente.

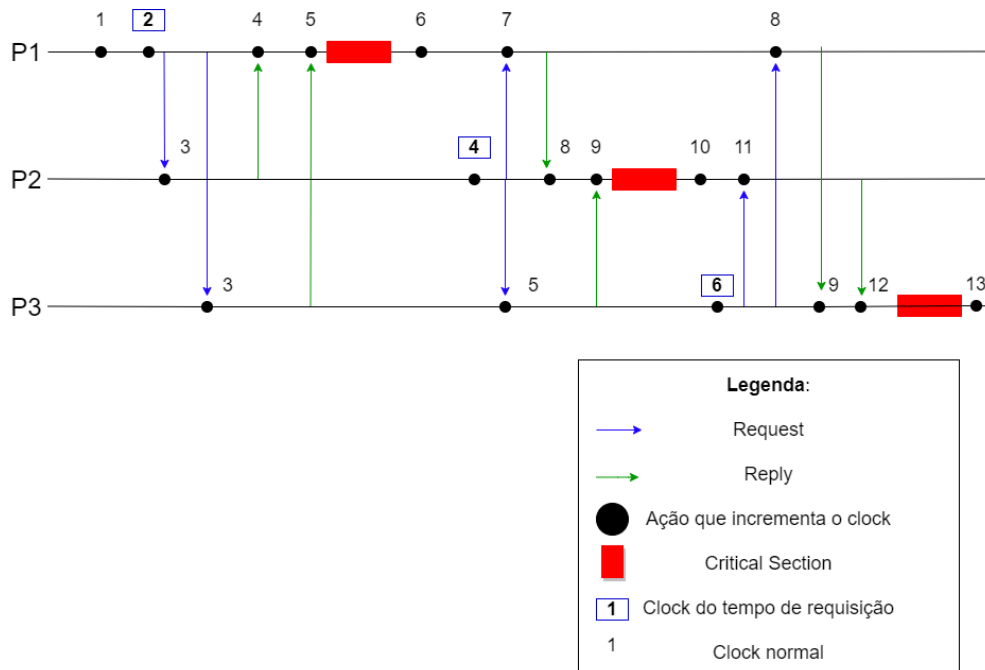


Figura 1: Exemplo de esquema testado