MC714: Sistemas Distribuídos

Instituto de Computação - UNICAMP

Prof. Luiz Fernando Bittencourt

2° Trabalho

Alunos:

- Gabriel Dourado Seabra - RA: 216213
- Lucas de Paula Soares - RA: 201867

Link para o projeto no GitHub:

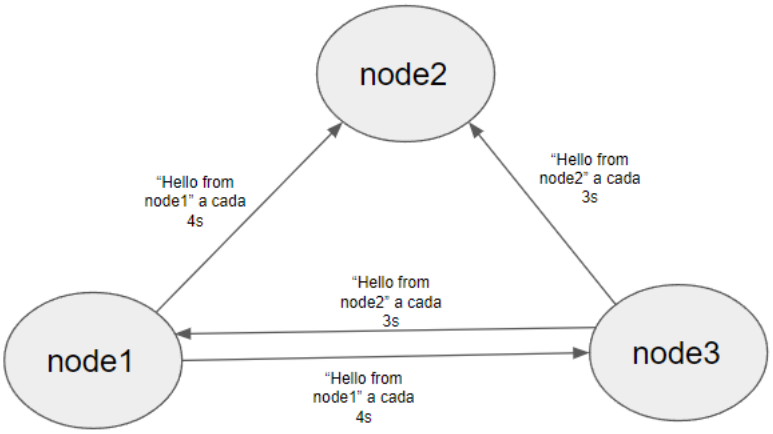
- https://github.com/lucasdpss/MC714-Sistemas-Distribuidos

Link para vídeo com explicação e demonstração da solução:

- https://youtu.be/DEUM-J7YcCE

Implementação da comunicação:

Foi feita utilizando Python e o message broker RabbitMQ, devido a sua facilidade de configurar e boa documentação com códigos iniciais disponíveis neste endereço: https://www.rabbitmq.com/tutorials/tutorial-one-python.html [1]. O servidor do message broker está rodando em uma máquina EC2 da Amazon, com seu IP e porta disponíveis para o serviço. Para testar todo o mecanismo desenvolvido, fizemos uma comunicação com três nós, o nó 1 envia uma mensagem para os nós 2 e 3 a cada 4s e o nó 3 envia uma mensagem para os nós 1 e 2 a cada 3s, enquanto o nó 2 apenas recebe as mensagens. Cada nó é executado em uma instância do Google Cloud. Segue o diagrama abaixo:



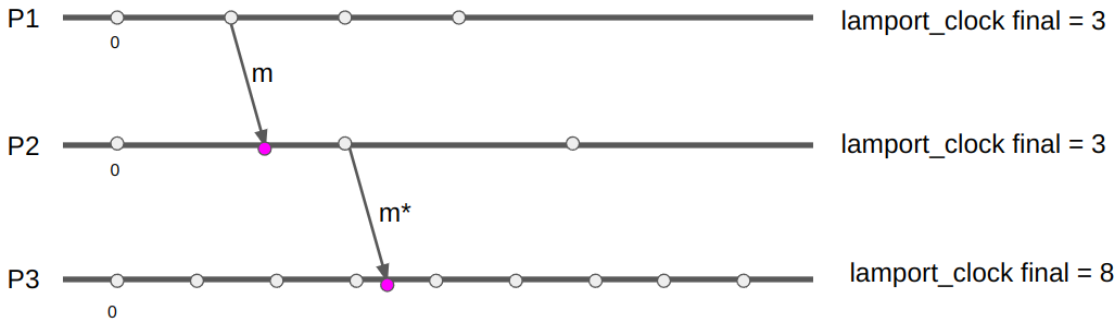
1) Relógio lógico de Lamport.

Nesse problema, precisamos sincronizar a ordem de eventos entre processos Pi, e para isso cada processo mantém um contador local Ci e esse contador muda em três situações diferentes:

1) Antes de executar um evento (enviar msg, entregar msg a uma aplicação, etc), Pi executa Ci <- Ci + 1

2) Quando Pi envia msg m para Pj: ts(m) <- Ci
3) Ao receber msg m, Pj faz Cj <- max{Cj , ts(m)}, executa (1) e entrega mensagem para aplicação.

Para implementar essa solução, criamos uma situação exemplo com três processos, como exemplificado abaixo:



O Processo 1 envia uma mensagem ao processo 2 e o processo 2 envia uma mensagem para o processo 3 contendo o seu relógio lógico atual. Para salvar o estado do relógio lógico precisamos usar thread locks, visto que temos duas threads que salvam em uma mesma variável “lamport_clock”, para que a thread de recebimento das mensagens não tenha um *race condition* com a thread principal do processo.

Ao rodar o programa em cada nó tivemos a seguinte situação em cada processo:

```
Mensagem enviada para P2:  1
Evento local: 2
Evento local: 3
lamport_clock final: 3
```

Logs do processo P1

```
Received: b'1'
Mensagem enviada para P3:  2
Evento local: 3
lamport_clock final: 3
```

Logs do processo P2

```
Evento local: 1
Evento local: 2
Evento local: 3
Received: b'2'
Evento local: 4
Evento local: 5
Evento local: 6
Evento local: 7
Evento local: 8
lamport_clock final: 8
```

Logs do processo P3

O Processo P3 tem vários eventos e aumenta rapidamente seu relógio lógico, e ao receber a mensagem de P2, seu relógio C3 estava C3=3, e como recebeu “2” pela comunicação, C3 permaneceu 3 e o próximo evento local foi para C3=4.

O Processo P2 começa recebendo a mensagem “1” de P1 e atualiza seu clock para C2=1, por isso quando envia a mensagem para P3, é enviado “2”.

E o Processo P1 inicia enviando uma mensagem para P2 e em seguida tem mais dois eventos locais que alteram seu clock.

O código foi feito em python e utilizamos os módulos *threading* para lidar com a thread principal e a thread de recebimento de mensagens de cada nó. Utilizamos também o módulo *time* para simular uma computação em um determinado tempo, através do método time.sleep(x), onde x é o tempo em segundos. E utilizamos o módulo *pika* que implementa a comunicação com o servidor RabbitMQ para comunicação[1], e é preciso instalar em cada máquina de cada nó através do comando: python -m pip install pika --upgrade

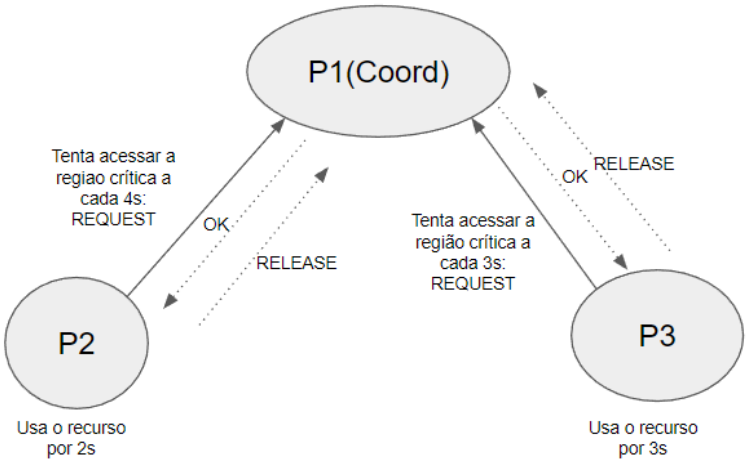
O código foi feito baseando-se nos slides da disciplina sobre sincronização e relógios lógicos de Lamport.

2) Algoritmo centralizado para exclusão mútua

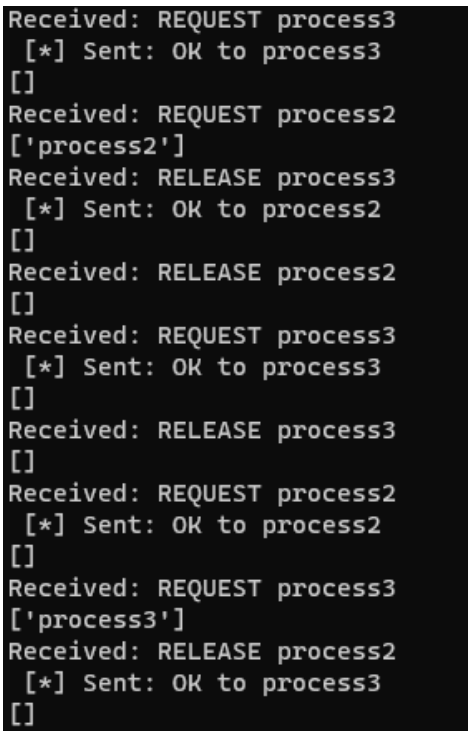
Este algoritmo precisa de um coordenador para forçar a exclusão mútua e assim cada nó possa realizar uma operação em uma região crítica. Existem três tipos de mensagem: “REQUEST”. “RELEASE” e “OK”. O nó que deseja acessar o recurso precisa pedir permissão para o coordenador com um “REQUEST” e este libera caso não tenha ninguém na fila com uma mensagem “OK”. Após o processo que pediu autorização utilizar o recurso este precisa enviar um “RELEASE” para o coordenador indicando que terminou de realizar a operação crítica.

Utilizamos a linguagem Python e as mesmas bibliotecas para comunicação do item anterior. O código foi feito baseado no pseudo-código apresentado nos slides da disciplina CIS 505: Software Systems, Lecture Note on Synchronization [2].

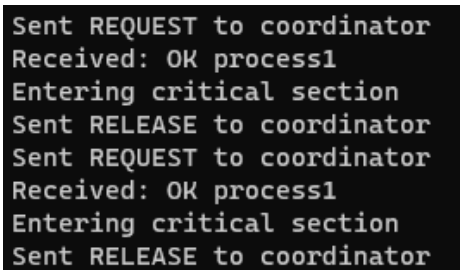
Para implementar esse cenário, criamos três nós, sendo o nó P1 o coordenador e os nós P2 e P3 os processos que concorrem com o recurso. O processo P2 tenta acessar a cada 4s a região crítica, e o utiliza por 2s, enquanto o processo P3 tenta acessar a região crítica a cada 3s e o utiliza por 1s.



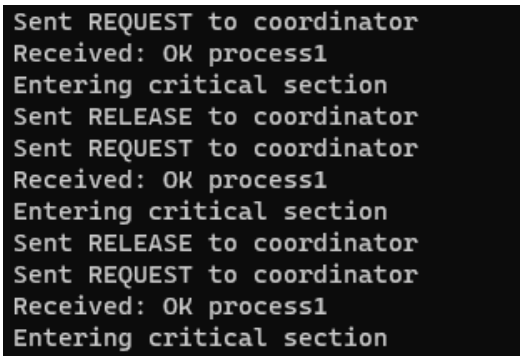
Ao rodar os códigos em cada nó, obtemos a seguinte situação em cada processo:



Logs do processo P1 coordenador



Logs do processo P2



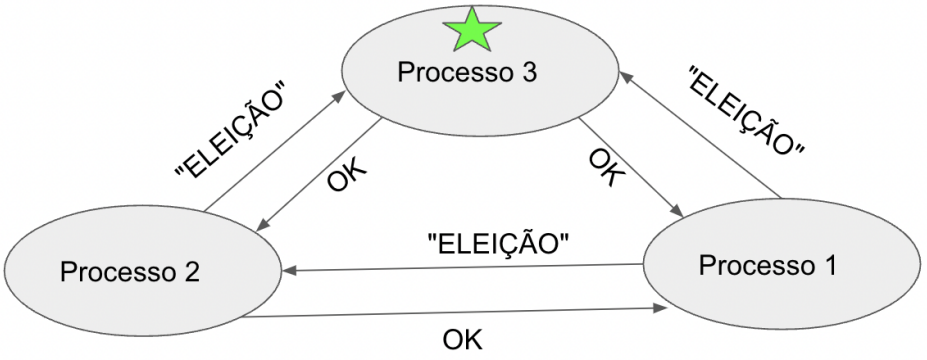
Logs do processo P3

O Processo coordenador coloca na tela qual mensagem recebeu e para quem enviou. No recorte analisado, o algoritmo garantiu a exclusão mútua. A desvantagem desse mecanismo é que o coordenador é o principal ponto de falha, e da forma como foi implementado, não é possível distinguir entre uma permissão negada de uma falha no coordenador.

3) Algoritmo de Eleição de Líder

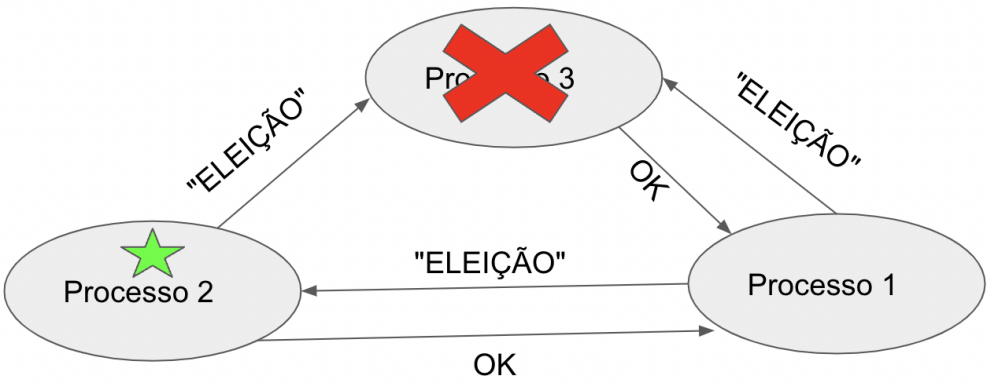
Usamos o Algoritmo de eleição de líder do valentão. Para isso, tivemos que partir da premissa de que cada processo conhece quais processos são maiores que ele, além de conhecer a sua própria ordem. No exemplo, usamos três processos numerados de 1 a 3. Cada processo, quando inicializado ou quando percebe que o coordenador está indisponível (isso é detectado através de sucessivas mensagens de "teste" a cada 2 segundos ao líder), envia uma mensagem de eleição para todos os nós superiores a ele. Se não obter nenhuma resposta "OK", ele se declara líder e propaga uma mensagem de "WINNER" para todos os nós menores. A figura a seguir resume o que ocorre quando todos os processos respondem com OK.

= Coordenador



Nesse caso, o líder é o processo 3 pois é o de maior valor. No caso em que o processo 3 não estiver respondendo, o processo 1 envia uma mensagem de eleição aos processos 2 e 3, no qual o processo 2 responde com um "OK" e o processo 3 não responde. Já o processo 2 envia uma mensagem de eleição apenas ao processo 3, que não responde. Como 3 é o único processo maior que 2, 2 assume a liderança e propaga uma mensagem de WINNER ao processo 1, que o aceita como novo coordenador. A Figura abaixo ilustra essa situação e o log exibe as mensagens trocadas na interface do processo 2 depois do nó 3 ser adicionado. Na metade da execução percebe-se o momento em que o nó 3 passa a não responder mais e o nó 2 assume a liderança.

= Coordenador



```
node2 sent node3 message: teste
node2 received message: OK from node3
lider: node3
node2 sent node3 message: teste
node2 received message: OK from node3
lider: node3
node2 sent node3 message: teste
node2 received message: OK from node3
lider: node3
node2 sent node3 message: teste
node2 sent node3 message: ELEICAO
node2 received message: ELEICAO from node1
node2 sent node1 message: OK
node2 sent node2 message: ELEICAO
node2: eu sou o líder eleito.
node2 sent node1 message: WINNER
node2 received message: teste from node1
lider: node2
node2 sent node1 message: OK
lider: node2
lider: node2
lider: node2
```

Logs do processo 2

Referências

[1] Python introduction for RabbitMQ Tutorials. Disponível em https://www.rabbitmq.com/tutorials/tutorial-one-python.html Acessado: 04/12/2022

[2] CIS 505: Software Systems, Lecture Note on Synchronization. Disponível em https://www.cis.upenn.edu/~lee/07cis505/Lec/lec-ch6-synch3-v2.pdf Acessado: 04/12/2022

[3] MC714: Sistemas Distribuídos , Prof. Luiz Fernando Bittencourt. Algoritmos de eleição Slides das aulas 17 a 19