Trabalho 02 de MC714
Sistemas Distribuídos
Prof. Luiz Fernando Bittencourt

Aline Hesley Silva Sousa, RA:248490
Stéfany Coura Coimbra, RA: 289812

1. Descrição do problema

O problema consiste em implementar relógios lógicos de *Lamport*, um algoritmo de exclusão mútua e outro de eleição de líder. Esses temas são essenciais para sistemas distribuídos, respectivamente, resolvem a ordenação de eventos, o acesso à recursos compartilhados de modo que apenas um processo acesse por vez o recurso e a eleição de um líder que vai coordenar/permitir o acesso a esses recursos.

2. Algoritmos escolhidos

2.1. Algoritmo de Lamport com Exclusão Mútua

O algoritmo implementado para resolver o problema de exclusão mútua em sistemas distribuídos utiliza relógios lógicos de *Lamport* para garantir a ordenação total dos eventos. Em sistemas distribuídos, onde processos independentes precisam acessar recursos compartilhados sem um coordenador central, a ordenação dos eventos se torna essencial para evitar conflitos e garantir que apenas um processo acesse o recurso por vez. A base do algoritmo é o conceito de "aconteceu antes" (*happened-before*), que define uma relação de ordem parcial entre eventos em diferentes processos. Se um *a* evento ocorrer antes de um evento *b* no mesmo processo, ou se *a* é o envio de uma mensagem e *b* é o recebimento dessa mensagem, então *a -> b*. Esta relação é estendida para múltiplos eventos, formando uma ordenação transitiva.

No contexto do algoritmo de exclusão mútua, cada processo mantém um relógio lógico que é incrementado em cada evento (local ou de comunicação). Quando um processo deseja acessar o recurso, ele envia uma mensagem de solicitação a todos os outros processos, incluindo seu timestamp atual. Os processos que recebem esta solicitação colocam-na em suas filas de requisições e respondem com uma mensagem de confirmação. Um processo só pode acessar o recurso quando sua própria solicitação é a mais antiga na fila (menor *timestamp* e, em caso de empate, menor *process-id*) e ele já recebeu confirmações de todos os outros processos para requisições com *timestamps* posteriores. Quando um processo termina de usar o recurso, ele envia uma mensagem de liberação a todos os outros processos, que então removem a solicitação correspondente de suas filas. Este método garante que a ordem de acesso ao recurso seja determinística e baseada no *timestamp*, prevenindo conflitos e garantindo que apenas um processo use o recurso de cada vez.
Isso funciona porque o relógio lógico de Lamport cria uma ordenação parcial dos eventos. Ao usar isso junto de um desempate por um id do servidor, conseguimos uma ordenação total.

Vale ressaltar que a condição de que todos os demais serviços também estão solicitando a zona crítica pode ser ineficiente em muitos casos, porém, fornece um acesso justo para os casos de acessos muito frequentes a uma região compartilhada, evitando *starvation*.

2.2. Algoritmo de Eleição de Líder

O algoritmo escolhido para a implementação de eleição de líder foi o de *bully* (algoritmo de valentão). Dado um sistema que requer um líder, tal algoritmo funciona como uma forma de eleger um novo líder se ocorrer alguma falha ou indisponibilidade no nó responsável atual. O funcionamento se dá da seguinte forma:

1) Um ou mais nós percebem a falha no líder atual.
2) Eles iniciam uma mensagem de multicast para os nós de maiores IDs que os deles.
3) Se os nós que recebem a mensagem respondem “OK”, quer dizer que realmente possuem um ID maior que o do remetente da mensagem e estão conectados ao sistema. Ou seja, têm possibilidade de serem o novo líder.
4) Eles mandam uma mensagem de *multicast* para as IDs maiores que o deles e, assim sucessivamente, até que um nó não recebe uma mensagem de “OK” e, assim, pode se autodeclarar como o novo líder.
5) O nó ou os nós coordenadores que iniciaram a eleição então mandam uma mensagem de *broadcast* para todos os nós divulgando o novo líder do sistema.

3. Implementação

3.1. Algoritmo de Lamport com Exclusão Mútua

Utilizamos *Docker* para criar um ambiente de containers na nossa máquina. O passo a passo para executar os containers e implementação do algoritmo, é dado por:

1) Construir a Imagem *Docker*:

docker build -t lamport-lock

2) Subir os *Containers* com *Docker Compose*:

docker compose up

3) Verificar os *Logs*: Para conferir os *logs* de um dos *peers* (por exemplo, *peer1*):

docker logs peer1

Temos três arquivos no repositório do *github*:

docker-compose.yaml: Este arquivo configura e define como os serviços *Docker* devem ser executados em conjunto. Ele define três serviços *(peer1, peer2, e peer3)*, que representam os diferentes processos do sistema distribuído, a rede em que operam, a imagem que rodam e também é passado para a imagem uma variável ambiente com o id do *peer*, que é usado para a porta do *socket*.
Dockerfile: define a imagem do docker, que consiste de um código em *python*.
lamport_lock.py: A implementação de fato do sistema. Nele, 3 threads alternam em pegar uma zona crítica 20 vezes, e são impressos mensagens de *log* para quando mensagens são recebidas ou quando a zona crítica é acessada.

3.2. Algoritmo de Eleição de Líder

Simulamos um sistema distribuído através da biblioteca ZMQ da linguagem *python*, onde utilizamos um mecanismo de *Publisher-Subscriber* com *Sockets* e diferentes portas a depender do ID do nó dentro do sistema. Além disso, construímos em código uma *thread* para cada nó, de tal maneira que as mensagens de *multicast* concorrentes não fossem bloqueadas e o algoritmo funcionasse normalmente conforme as premissas.

Há dois métodos principais no código:
leaderMessage(): responsável pelo tratamento das mensagens de eleição enviadas para os nós do sistema, criação de *sockets* e *threads* ouvintes dos processos.

responder(): método responsável pelo tratamento das mensagens recebidas pelos nós e pelos remetentes *multicast*.

3.3 Sistema de comunicação

No caso do algoritmo de Lamport com exclusão mútua, para a comunicação entre processos, escolhemos utilizar sockets AF_INET com o protocolo UDP (socket.SOCK_DGRAM). Esta abordagem permitiu a comunicação rápida e eficiente entre processos. Diferente do TCP, o UDP é um protocolo não orientado a conexão, o que significa que ele não garante a entrega das mensagens, mas é mais rápido e tem menos *overhead*.

Já em relação ao algoritmo de eleição de líder, foi utilizada uma comunicação via *socket* também com o TCP através do mecanismo de *publish-subscribe*, pela facilidade de implementação, testes e *debugs* do código.

3.4 Bibliotecas

As principais bibliotecas utilizadas no algoritmo lamport-lock foram:

socket: A biblioteca socket permite criar sockets AF_INET (IPv4) para enviar e receber mensagens UDP, facilitou a interação entre os diferentes containers *Docker*, que atuam como processos independentes no algoritmo. Funções como *socket.socket()*, *sock.sendto(), sock.bind()*, e *sock.recvfrom()* são usadas para configurar e manejar a comunicação via *sockets*.

json: A biblioteca json é usada para serializar e desserilizar mensagens em formato JSON. Esta foi útil para transmitir dados estruturados entre processos de forma padronizada e legível. As mensagens trocadas entre os processos incluem informações como o tipo de mensagem, *timestamp*, e pid, e são codificadas e decodificadas usando *json.dumps()* e *json.loads()*. No caso específico do algoritmo implementado as mensagens incluem tipos como *request, release,* e *reply*.

os: A biblioteca os é usada para interagir com variáveis de ambiente do sistema operacional. No contexto do código, os.environ é usado para obter o identificador do processo (PEER_ID) a partir das variáveis de ambiente definidas nos containers *Docker*.

A principal biblioteca utilizada no algoritmo de bully foram:

ZMQ: ZeroMQ (ou ØMQ, 0MQ, ZMQ) é uma biblioteca de mensagens assíncronas de alto desempenho que permite a comunicação entre processos (IPC), bem como a comunicação entre diferentes dispositivos e redes. Ela é conhecida por ser simples e eficiente, permitindo a troca de mensagens de forma rápida e com baixa latência. No contexto do código, foi usada para o mecanismo de criação de sockets com *Publish-Subscribe*.

Ambas as implementações utilizaram a biblioteca:

threading: Esta biblioteca fornece ferramentas para trabalhar com *threads* em *Python*. No primeiro algoritmo, a biblioteca é usada para criar e gerenciar *threads* que permitem a execução simultânea de processos independentes, representando processos distribuídos. A classe LamportClock usa um *threading.Lock* para garantir a atomicidade das operações no relógio lógico. A classe Process usa um *lock* para proteger o acesso a variáveis compartilhadas e threading.Thread para escutar mensagens em segundo plano. Por exemplo, *listener_thread* é uma *thread* que escuta mensagens de outros processos. No segundo caso, foi utilizada para criar threads separadas para os processos de envio de mensagens de eleição e para tratamento dos nós no sistema a fim de garantir simultaneidade nas mensagens de *multicast*.

4. Provas da solução de Lamport

A prova do algoritmo vem do célebre paper de *Lamport "Time, Clocks, and the Ordering of Events in a Distributed System"*, que define como obter uma ordenação total dos eventos em um sistema distribuído.
Por causa que existe a garantia que há essa ordenação total, os eventos são garantidos para acessar a zona crítica um de cada vez.
É possível ainda provar que um processo de id *i* visita pela *n*-esima vez a zona crítica apenas se todos os outros processos já visitaram a zona crítica pelo menos n-1 vezes. A prova vem do fato de todos os processos esperarem por uma requisição de todos os outros antes de pedirem pelo processo. Essa etapa cria uma espécie de "sincronização" dos relógios, de modo que se um processo requisitar novamente a zona crítica, o valor do seu relógio lógico será maior que dos processos que requisitaram mas ainda não acessaram. Isso fornece justiça ao algoritmo, pois não há *starvation*.

5. Resultados

5.1. Algoritmo de Lamport com Exclusão Mútua

Para visualizar a troca de mensagens bem como o acesso a zona crítica, adicionamos mensagens de log, as quais são mostradas abaixo:

Exemplo de output:

```
...
peer2  | Processo 2 retorna evento de 1
peer2  | Mensagem de 1 recebida por 2. Tipo da msg=reply
peer2  | Mensagem de 3 recebida por 2. Tipo da msg=request
peer1  | Mensagem de 3 recebida por 1. Tipo da msg=reply
peer3  | Mensagem de 2 recebida por 3. Tipo da msg=reply
peer2  | Processo 2 retorna evento de 3
peer2  | Mensagem de 3 recebida por 2. Tipo da msg=reply
peer2  | Mensagem de 1 recebida por 2. Tipo da msg=release
peer1  | Acesso a zona critica por 1 no tempo 09:18:22, pela 1-esima vez
peer1  | Lista de processos:  [(1, 1), (1, 2), (1, 3)]
peer2  | Mensagem de 1 recebida por 2. Tipo da msg=request
peer1  | Liberando zona critica. Lista:  [(1, 1), (1, 2), (1, 3)]
…
```

Como resultado, nós criamos um algoritmo de sistemas distribuídos usando o *locker* que implementa o relógio de *Lamport* e resolve o problema da exclusão mútua.

5.2. Algoritmo de Eleição de Líder

Uso do algoritmo:

python bullyAlgorithm.py numProcess numConnected numStarters

Sobre os argumentos:
--> numProcess: quantos nós existem no coinc
--> numConnected: quantos nós estão conectados no sistema
--> numStarters: quantos nós iniciam a eleição

Para numProcess = 10, numConnected = 4, numStarters = 1:

```
python bullyAlgorithm.py 10 4 1
Connected nodes:
[0, 7, 1, 9]
Election starters:
[0]
Check Node --> Pid 11420, Node Id: 0, Starts an election: True
Check Node --> Pid 12016, Node Id: 7, Starts an election: False
Check Node --> Pid 14908, Node Id: 1, Starts an election: False
Check Node --> Pid 3900, Node Id: 9, Starts an election: False
NODE 0 MULTICASTS LEADER MESSAGE
NODE 9 RESPONDS OK TO NODE 0
NODE 7 RESPONDS OK TO NODE 0
NODE 1 RESPONDS OK TO NODE 0
```

```
NODE 7 MULTICASTS LEADER MESSAGE
NODE 9 MULTICASTS LEADER MESSAGE
NODE 1 MULTICASTS LEADER MESSAGE
NODE 9 RESPONDS OK TO NODE 7
NODE 7 RESPONDS OK TO NODE 1
NODE 9 RESPONDS OK TO NODE 1
NEW LEADER: 9
```

Para numProcess = 10, numConnected = 6, numStarters = 2:

Python

```
python bullyAlgorithm.py 10 6 2
Connected nodes:
[3, 9, 2, 4, 1, 8]
Election starters:
[9, 4]
Check Node --> Pid 27360, Node Id: 3, Starts an election: False
Check Node --> Pid 24108, Node Id: 9, Starts an election: True
Check Node --> Pid 30156, Node Id: 4, Starts an election: True
Check Node --> Pid 4052, Node Id: 2, Starts an election: False
Check Node --> Pid 20564, Node Id: 1, Starts an election: False
Check Node --> Pid 22004, Node Id: 8, Starts an election: False
NODE 4 MULTICASTS LEADER MESSAGE
NODE 9 MULTICASTS LEADER MESSAGE
NODE 9 RESPONDS OK TO NODE 4
NODE 8 RESPONDS OK TO NODE 4
NODE 8 MULTICASTS LEADER MESSAGE
NODE 9 RESPONDS OK TO NODE 8
NEW LEADER: 9
```

6. Fontes

Stein, Connor. Lamport clocks in python. Mostly computers. 6 de maio de 2018. Disponível em: <https://connorwstein.github.io/Lamport-Clocks/>. Acesso em: 30 de junho de 2024.

Ari, Alper. Python implementation of Bully Election Algorithm using ZMQ sockets for multiprocessor communication. Github. Disponível em: <https://github.com/alperari/bully/tree/main>. Acesso em: 04 de julho de 2024.