Universidade Estadual de Campinas

MC714 - Sistemas Distribuídos

Prof. Dr. Luiz Fernando Bittencourt

Trabalho 2

Artur Alves Cavalcante de Barros - 248232

João Deltregia Martinelli - 248342

Introdução

A coordenação no ambiente de sistemas distribuídos é uma operação fundamental para o bom funcionamento e ordenação de múltiplos processos. Este trabalho aborda a implementação de três algoritmos essenciais para resolver problemas nesse contexto: o relógio lógico de Lamport, o algoritmo de eleição de líder *Bully* e o algoritmo de exclusão mútua Token Ring.

O relógio lógico de Lamport visa resolver o problema da sincronização de relógios, permitindo a ordenação consistente de eventos em sistemas distribuídos sem a necessidade de um relógio físico compartilhado, algo que é inviável em sistemas distribuídos. Algoritmos de eleição são cruciais para escolher um coordenador, por exemplo, que se responsabiliza pelo gerenciamento de recursos e processos. O algoritmo de eleição de líder Bully foi escolhido pela sua simplicidade conceitual e de implementação. Já o algoritmo de exclusão mútua Token Ring garante que apenas um processo acesse um recurso crítico por vez, utilizando um "token" que circula entre os processos, evitando condições de corrida e pontos únicos de falha, sendo eficiente em ambientes distribuídos.

Implementação dos Algoritmos

A implementação dos algoritmos foi feita utilizando Docker, ZeroMQ e Python. Múltiplos containers docker foram criados com seu próprio endereço executando os algoritmos. A biblioteca python ZeroMQ foi utilizada para gerenciar a comunicação, por meio de troca de mensagens, entre os containers.

Relógio de Lamport

Para o relógio lógico de Lamport, a implementação é razoavelmente simples. Da forma que implementamos, cada nó envia mensagens contendo o valor do relógio do remetente a cada outro nó na rede. Ao receber uma mensagem, o nó incrementa seu relógio baseado no maior valor entre o seu relógio ou o recebido. Para implementar isso, foram criadas duas threads por nó. Uma é responsável por receber as mensagens, e a outra é responsável por enviar os seus valores de relógio periodicamente. Ao início da execução, cada nó escreve no log o valor de relógio com que está inicializando. Além disso, os nós registram todos os momentos que enviam ou recebem valores de relógios de outros nós.

Para fins de teste, foi desenvolvido um script que gera automaticamente o docker-compose.yml com uma quantidade arbitrária de nós. Além disso, cada nó é inicializado com um inteiro aleatório entre 1 e 1000, para podermos observar o comportamento de diferentes condições de inicialização. Com isso, observamos nos logs que os nós inicializados com os menores valores de relógio são rapidamente atualizados para 1 + o maior valor de relógio na rede no momento, e então continuamente incrementados, como é esperado

Algoritmo de exclusão mútua: Token Ring

Escolhemos o algoritmo Token Ring para implementar exclusão mútua em sistemas distribuídos. Garantir que apenas um processo acesse uma seção crítica por vez é

essencial para evitar deadlocks, starvation, condições de corrida e garantir a corretude do código. Um exemplo em que esse mecanismo se faz necessário é quando mais de um nó precisa acessar e modificar uma variável compartilhada.

O algoritmo é bem simples e pode ser resumido da seguinte maneira: Os processos se organizam em um anel lógico, onde cada nó possui um atributo indicando quem é o próximo agente no círculo. Um token é passado entre os nós, e apenas quem tiver o token pode acessar os recursos na seção crítica. Esse algoritmo, portanto, é distribuído, uma vez que não necessita de um nó central que coordene os outros. Existem alguns complicantes entretanto. O token pode se perder, fazendo com que os nós não sejam capazes de acessar os recursos que desejam. Para mitigar esse problema, é necessário implementar algum mecanismo de tolerância a falhas. No nosso caso, foi implementado uma funcão *monitor_token* que monitora quanto tempo se passou desde a última vez que o nó recebeu o token, mandando um sinal para regeracão do mesmo se o tempo for muito grande.

A outra função principal, executada em uma thread própria de cada processo, é *receive_token*, que fica escutando se o token chega no nó. Quando chega, é gasto alguns segundos para simular o trabalho na sessão crítica e então o token é passado para o próximo processo no círculo.

O token ring foi implementado com sucesso, incluindo um mecanismo de tolerância a falhas. O sistema é capaz de escalar para um número arbitrário de nós, contando que a variável *token_timeout* seja atualizada. A parte mais difícil da implementação do algoritmo foi fazer corretamente as comunicações entre os processos. Entender como usar a biblioteca pyzmq, e criar uma rede usando docker-compose foi complicado. O algoritmo em si foi relativamente simples de implementar, uma vez que o conceito por trás foi bem explicado pelo professor.

Algoritmo de eleição: bully

Para um algoritmo de eleição de líder, o algoritmo *bully* ou valentão foi implementado. Cada nó criado recebe os seguintes argumentos:

- Seu endereço e id
- Uma variável booleana *start_election* que indica se esse nó irá iniciar uma eleição ao início, ou seja, se esse nó foi o primeiro que notou a queda do coordenador.
- Os endereços dos outros processos junto com seu id

Antes de executar a função main, foi implementado, para fins de teste, que cada processo (com exceção de processos com *start_election* = 1) tem 50% de chance de estar morto, ou seja, não irá executar o código, podendo assim simular um nó inativo. No início da main, todo processo executado registra no log que está vivo. Após isso, o programa se divide em duas threads: uma que é responsável apenas por receber as mensagens, e a outra que é responsável por convocar eleições. Quando um nó é designado com *start_election* = 1, ele inicia a eleição, mandando essa mensagem para todos os nós de id maior. Quando um nó recebe ELEIÇÃO, ele envia OK ao remetente e dispara, por meio de um evento de thread python, a execução da função de eleição, repetindo o processo. Quando um nó recebe OK, ele incrementa o valor de *received_ok*, inicializada com zero. Após enviar a mensagem de eleição para todos os nós de id maior (com espera máxima de 500 ms), ele avalia se o número de OKs recebidos é igual a zero e, caso seja o caso, envia mensagens avisando a todos os nós que ele será o novo líder. Ao receber a mensagem LIDER, o nó atualiza a variável *current_leader*. Uma variável booleana *has_started_election* é atualizada para evitar que um mesmo nó crie mais de uma eleição, tardando o processo

de eleição. Para fins de exibição do resultado, 15 segundos depois de uma thread descobrir seu novo líder, ela imprime o número do líder. Isso permite verificar facilmente que todos os nós vivos (registrados no início do log) exibem o mesmo líder ao final do processo.

Visto que cada nó é chamado por um serviço do docker-compose, arquivo que é tedioso alterar para testar com uma quantidade de nós diferentes, por exemplo, implementamos um script python que gera o arquivo. Isso nos permite facilmente testar o algoritmo de eleição com uma quantidade arbitrária de nós e escolher quais são os nós que se responsabilizaram pelo processo de votação inicial. Os testes foram implementados dessa forma, escolhendo possivelmente múltiplos nós “sementes”.