Trabalho 2

MC714 - Sistemas Distribuídos

Caio Petrucci dos Santos Rosa - RA 248245
Jonathan do Ouro - RA 248364

1 Introdução

Os algoritmos distribuídos são vitais para o funcionamento eficiente e confiável de sistemas onde múltiplos processos ou nós/instâncias precisam trabalhar juntos em alguma tarefa. Eles lidam com desafios complexos, como falhas de comunicação, concorrência e a necessidade de consenso.

Dentre os diversos problemas críticos de sistemas distribuídos, este trabalho englobou alguns deles: **marcação de tempo causal para ordenação de eventos**; **exclusão mútua para acesso a recursos compartilhados**; e **coordenação de tarefas centralizadas em sistemas com diversos nós**.

A ordenação de eventos em sistemas distribuídos é crucial para manter a sequência correta das operações. O Relógio de Lamport para ordenação de eventos oferece uma maneira lógica de ordenar eventos, mesmo sem um relógio global sincronizado, preservando a relação de causalidade entre eles.

Outro desafio fundamental é a exclusão mútua, que assegura que apenas um processo por vez acesse um recurso crítico, evitando conflitos e garantindo a integridade dos dados. O Relógio de Lamport para exclusão mútua é uma abordagem que utiliza carimbos de tempo lógicos para coordenar o acesso aos recursos, mantendo a ordem e a consistência.

Um dos problemas mais críticos em sistemas distribuídos é a eleição de líder, que garante que um nó específico assuma o papel de coordenador, gerenciando tarefas importantes e tomando decisões centralizadas. Um exemplo de algoritmo que soluciona uma parcela das instâncias desse problema é o algoritmo do valentão. Ele permite que os nós elejam um líder de forma dinâmica, mesmo na presença de falhas, garantindo que o sistema continue a operar de maneira eficiente.

Esses algoritmos são exemplos de como a complexidade dos sistemas distribuídos pode ser gerida de maneira eficiente e robusta, garantindo que os processos cooperem harmoniosamente e que os objetivos do sistema sejam alcançados.

2 Ambiente de desenvolvimento

Para implementar o sistema distribuído, utilizamos contêineres que simulam diferentes máquinas, executados na mesma rede para facilitar a comunicação entre os nós do sistema. Esses contêineres foram criados usando Docker. Um Dockerfile descreve como rodar nossos programas, e um arquivo docker-compose.yml cria várias instâncias desse mesmo contêiner.

Primeiramente, definimos um Dockerfile para criar a imagem do contêiner. Nesse arquivo, especificamos a imagem base do Python e copiamos o código-fonte do projeto para dentro do contêiner. Além disso, instalamos as dependências necessárias, como gRPCio, grpcio-tools e joblib, e geramos os arquivos Python relacionados com o *protobuf* do gRPC.

Para gerenciar e executar múltiplos contêineres de forma coordenada, utilizamos o Docker Compose, definido no arquivo docker-compose.yml. Este arquivo configura cinco serviços, cada um representando um nó no sistema distribuído, construindo a imagem Docker a partir do Dockerfile e definindo variáveis de ambiente específicas, como a porta do servidor e o ID do servidor. Essas variáveis distinguem cada instância do contêiner e permitem que se comuniquem entre si. Cada serviço roda em uma porta diferente, para não haver conflitos.

3 Relógio de Lamport

O Relógio de Lamport [1] é utilizado em sistemas distribuídos para manter a ordem dos eventos sem a necessidade de sincronização de relógios físicos. Neste trabalho, implementamos o relógio de Lamport como uma classe Python, onde a classe LamportClock é responsável por gerenciar o valor do relógio lógico.

A classe possui um método de inicialização que define o valor inicial do relógio como zero. O método tick incrementa o relógio a cada evento local. O método *update_clock* ajusta o relógio local ao receber mensagens de outros nós, garantindo a ordem dos eventos. O método *get_clock* retorna o valor atual do relógio.

4 Exclusão Mútua Distribuída

O algoritmo de exclusão mútua escolhido é o de Lamport, que mantém a ordem dos eventos do sistema com base no Relógio de Lamport. Ele funciona para

coordenar o acesso à recursos compartilhados entre nós de um sistema distribuido. Quando um nó do sistema deseja acessar a seção critica do programa ele monta uma mensagem que contem o recurso que ele deseja acessar, seu numero de processo e seu horario lógico (obtido com o relogio de Lamport), então envia essa mensagem para todos os nós do sistema e espera até receber confirmação de todos.

A nossa implementação utiliza comunicação gRPC, então definimos duas chamadas remotas para os programas, uma chamada é a *ReceiveRequestResourceUsage* que trafega o nome do recurso que deseja-se acessar, o *timestamp* de *Lamport* e o Id do nó que está pedindo acesso, a outra é a *ReceiveOkMessage* que recebe uma string generica que indica uma confirmação de uso de um recurso.

```
syntax = "proto3";

package myservice;

service NodeCommunicationService {
  rpc ReceiveRequestResourceUsage (UsageRequest) returns (UsageResponse) {}
  rpc ReceiveOkMessage (OkMessage) returns (UsageResponse) {}
}

message UsageRequest {
  int64 key = 1;
  int64 lamport_timestamp = 2;
  int64 server_id = 3;
}

message UsageResponse {
  string response = 1;
}

message OkMessage {
  int64 from_server_id = 1;
  string response = 2;
}
```

Listing 1: *Definição do serviço de comunicação RPC para o algoritmo distríbuido de exclusão mútua com Relógios de Lamport*

Para implementar essa comunicação descrevemos as chamadas remotas em um arquivo *.proto* e geramos os arquivos em python que implementam efetivamente a comunicação, para isso utilizamos a biblioteca *grpc_tools*. Nestes aquivos o serviço de comunicação que definimos no arquivo .proto se transforma em uma classe que pode ser extendida para personalizar a maneira de lidar com as requisições.

Criamos uma classe chamada *Storage* que extende o serviço de comunicação definido, nela definimos o método *set_value* que insere um valor em um dicionário compartilhado pelo sistema distribuido. Quando um nó deseja utilizar esse método ele pede permissão para todos os nós do sistema, isso através do método chamado *request_resource_usage* que cria conexões com os nós do sistema e chama o procedimento remoto *ReceiveRequestResourceUsage* de cada um.

O procedimento remoto *ReceiveRequestResourceUsage* implementa a lógica do algorimto de Lamport, se o nó estiver na seção critica ele não responde e enfileira o pedido, caso não tiver na seção critica e não quiser acesar ela o nó responde com um OK através do procedimento remoto *ReceiveOkMessage*, se o nó quiser acesso a seção critica ele compara seu timestamp de Lamport com o timestamp do nó que está requisitando acessar a mesma seção critica e da prioridade para pedidos de acesso que aconteceram antes.

Não obstante, o método *request_resource_usage* espera um tempo até que ele identifique que recebeu uma quantidade suficiente de OKs dos outros nós, que na nossa demonstração são 4 oks porque o sistema possui 5 instâncias.

O recurso compartilhado é um arquivo *.pkl* que armazena o dicionario alterado pelos nós do sistema. Para ler e salvar no arquivo utilizamos uma biblioteca chamada *joblib* que permite serializar objetos python e salvar em disco.

Para desenvolver o algoritmo de Lamport foi utilizado como base nos slides de aula [2]. E para mais detalhes de implementação, é possível consultar o código-fonte feito pela dupla, no repositório GitHub em `https://github.com/caiopetruccirosa/trabalho2-mc714-2024s1/`.

5 Eleição de Líder

O problema de eleição de líder em sistemas distribuídos refere-se à necessidade de escolher um único nó, ou processo, que atuará como coordenador ou líder para gerenciar determinadas funções críticas no sistema. Em ambientes distribuídos, onde não há um ponto central de controle, a eleição de um líder é crucial para assegurar que operações importantes sejam realizadas de maneira organizada e eficiente. O líder é responsável por tarefas como alocação de recursos, sincronização de processos e resolução de conflitos.

Também, é interessante observar que entre alguns problemas relacionados ao problema de eleição de líder está o problema de consenso. Enquanto a eleição de líder lida com a escolha de um coordenador, o problema de consenso se refere à necessidade de todos os nós em um sistema distribuído concordarem sobre um determinado valor ou estado. O consenso é fundamental para garantir que todos os nós tenham uma visão consistente do estado do sistema, o que é crítico para a integridade e a coerência das operações distribuídas.

Neste trabalho, implementamos o **algoritmo do valentão**, ou o ***bully algorithm***, que é uma das técnicas utilizadas para a eleição de um líder em sistemas distribuídos [3]. Este algoritmo assume que todos os nós em um sistema têm identificadores únicos e que

cada nó conhece os identificadores dos outros nós. O nome "valentão"se refere ao comportamento dos nós com identificadores mais altos, que "intimidam"os nós com identificadores mais baixos para assumir a posição de líder.

Em sua execução, o ***bully algorithm*** utiliza os seguintes tipos de mensagem:

- **Mensagem de Eleição**: Enviada para anunciar a eleição.
- **Mensagem de Resposta**: Responde à mensagem de Eleição.
- **Mensagem de Coordenador**: Enviada pelo vencedor da eleição para anunciar a vitória.

Adicionalmente, em nossa implementação, incorporamos uma **Mensagem de *Status Check***, de forma que os nós do sistema sejam capazes de checar se o seu líder atual ainda está em funcionamento.

Em sua estrutura geral, o algoritmo segue os seguintes passos:

1. Quando um processo P se recupera de uma falha, ou detecta que o coordenador atual falhou – na nossa implementação isso acontece através de envios periódicos de **Mensagens de *Status Check*** –, P realiza as seguintes ações:
   (a) Se P tem o maior ID de processo, ele envia uma **Mensagem de Vitória** para todos os outros processos e se torna o novo Coordenador. Caso contrário, P transmite uma **Mensagem de Eleição** para todos os outros processos com IDs de processo mais altos do que o seu.
   (b) Se P não recebe Resposta após enviar uma **Mensagem de Eleição**, então ele transmite uma mensagem de Vitória para todos os outros processos e se torna o Coordenador.
   (c) Se P recebe uma Resposta de um processo com um ID mais alto, ele não envia mais mensagens para essa eleição e aguarda uma **Mensagem de Vitória**. Se não houver **Mensagem de Vitória** após um período de tempo, ele reinicia o processo desde o início.
   (d) Se P recebe uma **Mensagem de Eleição** de outro processo com um ID mais baixo, ele envia uma **Mensagem de Resposta** e, se ainda não tiver iniciado uma eleição, inicia o processo de eleição desde o início, enviando uma **Mensagem de Eleição** para processos com ID mais altos.
   (e) Se P recebe uma **Mensagem de Coordenador**, ele trata o remetente como o coordenador.
2. Volte para o Passo 1 e execute em *loop*.

Em nosso trabalho, utilizamos a linguagem Python para a implementação dos nós, seguindo um funcionamente *Peer-2-Peer* (P2P), e, assim como para os outros algoritmos, utilizamos gRPC para a comunicação entre instâncias. Os métodos gRPC declarados em nosso serviço `.proto` estão descritos na Listing 2.

```
syntax = "proto3";

package node_service;

service Node {
  rpc RunElection (NodeRequest) returns (NodeResponse) {}
  rpc ReceiveCoordinatorMessage (NodeRequest) returns (NodeResponse) {}
  rpc GetCoordinatorStatus (NodeRequest) returns (NodeResponse) {}
}

message NodeRequest {
  int32 node_id = 1;
}

message NodeResponse {}
```

Listing 2: *Definição do serviço de comunicação gRPC para o algoritmo do valentão*

Nestas especificações, o método *RunElection* é responsável por desempenhar o papel tanto da **Mensagem de Eleição** quanto da **Mensagem de Resposta**, no processo de chamada e resposta de um procedimento remoto. Já a **Mensagem de Vitória** é representada pelo procedimento *ReceiveCoordinatorMessage*, que é chamado pelo nó líder após declarar sua eleição. Por fim, o método *GetCoordinatorStatus* é responsável pelas **Mensagens de *Status Check***, que indicam se o nó líder ainda está operacional e saudável.

Durante o desenvolvimento do trabalho, os materiais utilizados como base teórica foram os slides da disciplina MC714 [4] e a página da Wikipedia do algoritmo do valentão [3]. Para mais detalhes de implementação, é possível consultar o código-fonte feito pela dupla, no repositório GitHub em `https://github.com/caiopetruccirosa/trabalho2-mc714-2024s1/`.

6 Conclusão

Neste trabalho, implementamos dois algoritmos importantes para sistemas distribuídos: o algoritmo de exclusão mútua de Lamport e o algoritmo de eleição de líder Bully. A implementação desses algoritmos em um ambiente distribuído, utilizando Docker para simular múltiplas instâncias de máquinas se mostrou uma boa forma para vizualizar o funcionamento dos algoritmos.

O algoritmo de exclusão mútua de Lamport, através do uso de relógios lógicos, permite uma coordenação ordenada do acesso a recursos compartilhados. Em um cenário real, como um sistema de banco de dados distribuído, este algoritmo pode ser crucial

para garantir que múltiplas transações não entrem em conflito ao tentar modificar os mesmos dados simultaneamente.

O algoritmo do valentão, que serve para eleição de líderes em sistemas distribuídos, lida bem com a falha de nós e garante que o nó com o maior identificador sempre se torne o líder, evitando conflitos e mantendo a operação contínua do sistema. Em um ambiente real, a eleição de líder pode ser aplicada na coordenação de servidores em um cluster de banco de dados, onde um servidor principal precisa gerenciar a replicação de dados e a coordenação das transações.

Referências

[1] Wikipedia. *Relógios de Lamport — Wikipedia, The Free Encyclopedia*. http://pt.wikipedia.org/w/index.php?title=Rel%C3%B3gios%20de%20Lamport&oldid=58229253. [Online; Acessado em 01/07/2024]. 2024.

[2] Prof. Luiz Fernando Bittencourt. *MC714 - Sistemas Distribuídos, 1° semestre, 2024*. Slides, Slide 17 - p. 11-13, Instituto de Computação.

[3] Wikipedia. *Bully algorithm — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Bully%20algorithm&oldid=1132111922. [Online; Acessado em 02/07/2024]. 2024.

[4] Prof. Luiz Fernando Bittencourt. *MC714 - Sistemas Distribuídos, 1° semestre, 2024*. Slides, Slide 18 - p. 6-8, Instituto de Computação.

[5] *Introduction to gRPC | gRPC*. https://grpc.io/docs/what-is-grpc/introduction/. Acessado em 29/06/2024.

[6] *Quick start | Python | gRPC*. https://grpc.io/docs/languages/python/quickstart/. Acessado em 29/06/2024.

[7] *Basics tutorial | Python | gRPC*. https://grpc.io/docs/languages/python/basics/. Acessado em 29/06/2024.

[8] A.S. Tanenbaum e M. van Steen. *Distributed Systems*. CreateSpace Independent Publishing Platform, 2017. ISBN: 9781543057386. URL: https://books.google.com.br/books?id=c77GAQAACAAJ.