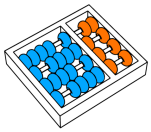
Universidade Estadual de Campinas
Instituto de Computação



MC714 - Sistemas Distribuídos
2º Trabalho

Integrantes: Christian Massao Konishi – 214570; Felipe Hideki Matoba – 196767
Link para Vídeo: `https://youtu.be/at3ONLUW6eA`
Link para Repositório: `https://github.com/ChrisKonishi/MC714_P2`

1 Introdução

Quando comparado a um sistema de processador único, um sistema distribuído pode apresentar inúmeras vantagens em questão de escalabilidade, disponibilidade, confiabilidade segurança etc., contudo programar nesse ambiente é consideravelmente mais complicado. Tipicamente os processadores em um sistema distribuído não compartilham um relógio físico, isto é, sincronizações absolutas baseadas em tempo não são confiáveis, assim como determinar a ordem com que os eventos ocorrem dentro do sistema, baseando-se somente em marcas de tempo.

Uma alternativa possível para esse problema é o uso de relógios lógicos. Entre os algoritmos possíveis, o Relógio de Lamport é uma opção que não é capaz de dizer a ordem de todos os eventos, mas é capaz de indicar nos casos em que há uma possível relação de causalidade entre eles. Neste trabalho, esse algoritmo foi implementado e discutido mais a frente (Seção 2).

Outro desafio que surge nesse cenário é o uso concorrente de recursos, se ele não puder ser manipulado por dois processadores ao mesmo tempo, se faz necessário a utilização de um algoritmo de exclusão mútua. A abordagem escolhida nesse projeto é a de um algoritmo centralizado, na qual um dos componentes — coordenador — da rede é responsável por dar permissão ou negar o acesso de outros ao recurso designado (Seção 3).

Por fim, pensando no caso anterior em que há um coordenador, definir e fazer com que todos os processadores entrem em acordo a respeito de quem é o líder é um outro desafio necessário de ser abordado. Note que quando há a presença de um líder, ele passa a ser um ponto crítico do sistema, que quando apresenta uma falha, deve ser tratado. Algoritmos de

eleição são uma possibilidade para se recuperar desses casos, em especial, um algoritmo de anel foi empregado no trabalho (Seção 4).

A implementação dos algoritmos para sistemas distribuídos foi feita utilizando trocas de mensagens pela rede. O uso de máquinas físicas propriamente separadas ou mesmo de máquinas virtuais em ambiente *cloud* não foi empregado. No lugar disso, os elementos pertencentes ao sistema distribuído foram isolados utilizando um serviço de containers — Docker [2] —, orquestrados pela ferramenta docker-compose, com comunicação pela rede *overlay* do Docker [4]. A comunicação entre containers foi feita através de requisições HTTP, sendo que o *framework* Flask [5] foi utilizado para criar os servidores web, e a biblioteca Requests [3], para construir e realizar as requisições, os dados trocados foram serializados em JSON. Os servidores HTTP foram executados utilizando o servidor Gunicon [1], com um processo e cinco *threads*. Não foram utilizados códigos de terceiros para a implementação dos algoritmos em si, o uso das bibliotecas para envio de mensagens e do Docker, devidamente referenciados, também eram de conhecimento prévio dos integrantes.

2 Relógio lógico de Lamport

Relógios lógicos têm o objetivo de estabelecer uma relação de ordem entre eventos de diferentes processos que comunicam-se entre si, sem a preocupação de estabelecer uma hora real. Sendo assim, Lamport propôs um modelo no qual mensagens que são enviadas contém o tempo de seu envio, o que permite que o processo que a receba possa compará-lo com o seu tempo local. Caso o instante contido na mensagem seja posterior ao seu próprio relógio, o processo o ajusta para que fique consistente com a ordem de acontecimento dos eventos.

Note que se o relógio de Lamport for utilizado para determinar a ordem de todos os eventos em um sistema distribuído, o resultado não será necessariamente correto, o algoritmo apenas garante a ordem de eventos que podem ter alguma relação de causalidade detectada, o que envolve eventos em sequência de um mesmo processo, ou a troca de mensagens entre dois ou mais, já que os eventos que precedem o envio da mensagem podem ter alguma relação de causalidade com os posteriores ao recebimento.

Para demonstrar a implementação são criados três processos, cada um correspondendo a um servidor HTTP, com IDs de 1 a 3. O relógio local de cada um é incrementado pelo número de seu ID a cada um segundo, o que cria uma discrepância entre seus horários. A tarefa a ser realizada, criada por uma aplicação também implementada como um servidor HTTP, consiste em uma string de números que indicam para qual processo ela deve ser enviada.

Inicialmente, a aplicação envia a tarefa para o processo do primeiro número da string por meio do *endpoint* /receive_msg, colocando a *timestamp* igual a 1. Esse processo recebe a mensagem e compara a *timestamp* com seu próprio relógio, ajustando-o para $timestamp + 1$ caso ela seja maior. Em seguida, espera dois segundos e pega o próximo número na string,

que contém o ID do próximo processo a ser chamado, e envia o resto da string junto com o instante marcado em seu relógio naquele momento. A lógica descrita se repete até que toda a string seja percorrida, então o último processo envia uma mensagem para a aplicação no *endpoint* /receive_answer, sinalizando que a tarefa foi cumprida.

Uma alternativa ao relógio lógico de Lamport é um relógio vetorial, no qual cada evento ocorrido em um processo incrementa seu relógio, cujo valor é adicionado ao vetor quando há comunicação com outro processo. Com isso, cada processo fica ciente de quantos eventos ocorreram em cada processo antes da mensagem ser enviada, adicionando também a noção de causalidade para chegar no ponto em questão.

3 Exclução Mútua

Em um algoritmo de exclusão mútua centralizado, é necessário que haja um coordenador, todos os processadores que precisam acessar um recurso protegido, isto é, cuja concorrência não é possível, devem enviar uma mensagem de requisição de acesso; o coordenador deve então checar se o recurso está disponível, se estiver, devolve uma mensagem de concessão. Caso contrário, o coordenador aguarda uma mensagem de liberação, disponibilizando o recurso ao próximo requisitante, em ordem.

Na prática, o algoritmo foi implementado com algumas simplificações, o coordenador corresponde a um servidor HTTP que contém o próprio recurso a ser acessado pelos outros clientes, no caso, um simples valor inteiro que pode ser lido ou escrito através de HTTP *requests*. Na demonstração, dois clientes tentam realizar uma série de operações no recurso, sempre primeiro requisitando o acesso no *endpoint* /request_access e liberando o recurso no final, /release_access, ambos com método GET.

Nessa implementação, o coordenador utiliza um semáforo com contador 1, para garantir que haja a exclusão mútua dos serviços, se ele não estiver disponível, o *request* fica bloqueado. A própria resposta do *endpoint* age então como a mensagem de concessão. Note então que se o cliente não quiser ficar bloqueado enquanto aguarda o recurso, deve requisitá-lo numa *thread* separada. Por fim, o uso do semáforo simplifica muito a implementação, mas acaba por não garantir a ordem dos recursos requisitando o acesso. A próxima concessão é dada para aquele que primeiro "adquirir"o semáforo. Essa adaptação foi julgada como aceitável durante a implementação, visto que o risco de *starvation* nessa pequena demonstração é nulo.

Por fim, os endpoints /get_resource (GET) e /change_resource (POST) permitem a manipulação do recurso protegido.

Existem outras abordagens para lidar com o problema. Um algoritmo descentralizado que utiliza uma DHT com n coordenadores é possível, nesse caso, é necessário requisitar acesso a todos os coordenadores, sendo liberado se houver aprovação de uma maioria. Ele é mais

tolerante a falhas, visto que na abordagem centralizada, uma falha no coordenador é capaz de gerar um defeito no sistema. Também é possível circular entre os processadores um token que libera acesso ao recurso, mas perder o token também é uma falha possível, e inúmeras mensagens circulando o *token* são trocadas aumentando a carga na rede. Existem também algoritmos distribuídos, no qual o cliente que quer acessar um recurso deve requisitar acesso a todos os outros, que passam a ser pontos possíveis de falha, além de aumentar a carga na rede.

Já no algoritmo centralizado, menos mensagens são trocadas, e o ponto de falha mais crítico passa a ser o coordenador. Se ele ficar indisponível, uma possível alternativa seria iniciar uma eleição, indicando o novo coordenador, o processador que estava com acesso ao recurso pode também ser informado junto às mensagens da eleição. Agora se o processador com acesso garantido falhar, ele nunca liberará o acesso, para isso, um *health check* pode ser implementado, e, na ausência do processador com defeito, o recurso se torna disponível novamente.

4 Algoritmo de Eleição de Anel

Um algoritmo de eleição pode ser utilizado em um sistema distribuído para determinar um líder, que pode agir como um coordenador em um algoritmo de exclusão mútua, por exemplo. Dentro de um sistema distribuído, o processo saudável de maior ID deve ser o vencedor de uma eleição.

O algoritmo implementado é o de anel, no qual os processos se conectam de forma circular, cada um se comunicando com seu sucessor. Quando o processo líder fica inativo, isso deve ser detectado, e um processo deve iniciar a eleição. A mensagem de eleição deve então circular a rede, cada processo repassando para seu sucessor e adicionando na mensagem o próprio ID. Se um processo sucessor está inativo, a mensagem deve ser enviada ao próximo sucessivamente.

Eventualmente a mensagem de eleição deve retornar a quem iniciou o processo, nesse momento, a eleição é finalizada e a mensagem contém todos os clientes ativos. O ID mais alto será o novo líder. Finalmente, o resultado da eleição e os IDs ativos são circulados novamente pela rede por meio de outra mensagem, informando todos do novo líder.

Na prática, o algoritmo foi implementado utilizando n containers formando um anel. Cada um deles contendo um servidor HTTP para recepção de mensagens e uma thread rodando concorrentemente, observando periodicamente o estado de saúde do processo sucessor, através do *endpoint* /health_check (GET). Se um request não retornar um status 200, é iniciado uma eleição através de um POST request em /start_election. Esse endpoint é responsável por requisitar a eleição no próximo processo, repassando o ID. Se houver um pedido de eleição e o ID do próprio processo estiver na mensagem, ele assume que a eleição foi iniciado

por ele mesmo e acabou de ser concluída. O resultado da eleição é então propagado aatravés de /inform_results (POST). Para simular o comportamento de um processo não saudável, é possível enviar um GET request a /kill, depois disso, todos os requests feitos contra esse servidor retornarão o status 500, indicando que o container não está mais saudável.

Todo container conhece a rede inteira, pois tem a informação da quantidade de containers n e sabe que os IDs obrigatoriamente variam de $1 \ldots n$, sendo que a URL de um container de id k é dada por *http://service<k>*

Outras alternativas de algoritmos envolvem o "do valentão", no qual cada processo envia mensagens de eleição para todos os IDs superiores, que se responderem devem repetir o mesmo processo, até restar apenas o de maior ID. Essa abordagem pode envolver um maior número de mensagens. Existem também algoritmos para rede sem fio, que envolvem a criação de uma árvore durante a eleição, selecionando um líder segundo algum critério específico, tal como nível de bateria do dispositivo.

5 Conclusão

Neste trabalho foram implementados com sucesso algoritmos importantes para coordenar o funcionamento de sistemas distribuídos, resolvendo questões que surgem quando se tem diversos computadores trabalhando em conjunto e comunicando-se, como o controle a acesso de recursos compartilhados e determinação da ordem de eventos.

Mais especificamente, o relógio lógico de Lamport oferece uma maneira fácil de ordenar eventos que ocorrem entre processos distintos mas que se comunicam, necessitando apenas de comparações e ajustes simples de relógios, o que evita adição de complexidade desnecessária ao sistema. Enquanto isso, a exclusão mútua centralizada poderia ser usada em conjunto com o algoritmo de eleição em anel, garantindo a existência de um coordenador e, portanto, o acesso correto a recursos compartilhados.

Também poderiam ser explorados outros algoritmos já mencionados com as mesmas funções, mas abordagens diferentes, como um relógio vetorial para a questão do ordenamento de eventos, um algoritmo descentralizado para garantir a exclusão mútua ou o algoritmo do valentão para eleição.

Referências

[1] *Gunicorn - WSGI server — Gunicorn 20.1.0 documentation*. URL: `https://docs.gunicorn.org/en/stable/` (acedido em 02/12/2022).

[2] Dirk Merkel. «Docker: Lightweight Linux Containers for Consistent Development and Deployment». Em: *Linux J.* 2014.239 (mar. de 2014). ISSN: 1075-3583.

[3] *Requests: HTTP for Humans — Requests 2.28.1 documentation*. URL: https://requests.readthedocs.io/en/latest/ (acedido em 02/12/2022).

[4] *Use overlay networks*. Dez. de 2022. URL: https://docs.docker.com/network/overlay/.

[5] *Welcome to Flask — Flask Documentation (2.2.x)*. URL: https://flask.palletsprojects.com/en/2.2.x/ (acedido em 02/12/2022).