Relatório trabalho 2 MC714 - Sistemas distribuídos

Felipe Martins Romeiro - 215720
Vinícius Waki Teles - 257390

Montagem de ambiente

Para montar um ambiente de sistemas distribuídos, optamos por usar imagens Docker. Para isso, criamos um Dockerfile que usa uma imagem Docker com Python 3.10 como base, copia os arquivos da pasta para a imagem Docker e executa o arquivo app.py. Então, montamos um arquivo docker-compose que sobe 4 cópias dessa imagem, possibilitando um sistema distribuído com 4 instâncias.

Para rodar os testes, primeiro execute `docker compose build` para montar as imagens dos containers de teste. Depois, basta rodar `docker compose up -e DS_TEST=<TESTE>` para rodar o teste de um dos algoritmos, onde `<TESTE>` pode ser `CLOCK`, `LEADER` ou `MUTEX`.

Relógio Lógico

Para testar o relógio lógico, criamos uma aplicação que sobe um servidor RPC que responde um número aleatório. Para implementar o relógio lógico, encapsulamos as chamadas do RPC, tanto de solicitação quanto de resposta, com chamadas ao relógio lógico.

Implementação

RPC

A implementação do servidor RPC foi feita no arquivo *app.py* usando a biblioteca `xmlrpc` por ser uma biblioteca padrão do Python, facilitando a montagem do ambiente. Para isso, nos baseamos no exemplo do PythonBrasil e na documentação oficial do Python.

A comunicação RPC foi fácil de implementar, a interface da biblioteca é bem explícita. Tivemos um único problema em que estávamos criando o servidor usando o endereço `"localhost"`, e por isso ele não estava visível para os demais serviços Docker. Porém, seguimos essa resposta do StackOverflow, substituímos o endereço por `"0.0.0.0"` e conseguimos conectar cliente e servidor RPC sem problemas.

Para comunicar os serviços Docker, simplesmente usamos o nome declarado no *docker-compose.yml*, como explicado nessa resposta do StackOverflow, e o mapeamento entre IP e DNS foi realizado automaticamente pelo Docker.

Visando o compartilhamento de relógio entre cliente e servidor RPC, fizemos as duas aplicações rodarem no mesmo processo, usando a API de threads do Python. Para isso, fizemos o servidor rodar numa thread própria e deixamos o cliente rodando na thread principal. Para implementar as threads, nos baseamos nesse artigo do Real Python. Sobre as threads, tomamos o cuidado de criar a thread do servidor como um *daemon* para que a instância não espere o servidor encerrar para fechar o processo, já que o processo do servidor é infinito.

Relógio Lógico

Para implementar o relógio lógico, criamos uma classe Python que mantém o estado do relógio. Essa classe implementa um relógio lógico de Lamport simples, ou seja, não vetorial.

Pensando no caso de eventos que não envolvem a comunicação entre processos, criamos um método `event` que simplesmente atualiza o relógio.

Já visando a comunicação entre os processos, criamos os métodos `send` e `recv` que recebem uma função, encapsulam a comunicação com as atualizações e comunicações de relógio necessárias e retornam uma função pronta para atualizar o relógio com a mesma interface que a original.

A implementação desses métodos foi baseada no algoritmo do Slide 10 da aula 17.

Paralelismo

Durante os testes, percebemos que primeiro várias requisições RPC ocorriam e depois o log de todas as requisições aparecia da uma vez. Pensamos que isso poderia ser um problema de threads, então começamos a usar implementações *thread safe*. Nos baseando nesse artigo do Real Python, substituímos os usos de `sleep` por `event.wait`. Nesse mesmo artigo, percebemos que `print` não é *thread safe*, então substituímos o seu uso pela biblioteca de logging, nos baseando em sua documentação.

Não observamos nenhum caso, mas percebemos que poderiam haver *race conditions* no relógio entre cliente e servidor, então encapsulamos os acessos à variável de relógio com um semáforo, nos baseando no artigo de threading do Real Python.

Testes

Para testar o funcionamento do relógio lógico, fizemos o cliente esperar um tempo aleatório e fazer uma chamada RPC para alguma outra instância aleatória. Com isso, conseguimos observar o funcionamento do relógio lógico a partir dos logs da classe `Clock`.

O comportamento aleatório do cliente viabilizou a criação de vários casos, permitindo averiguar o funcionamento do relógio em cada um deles.

Para executar as instâncias de teste com os logs, basta rodar `docker compose up --build`.

Eleição de Líder

Implementação

Para o sistema de eleição de líder, implementamos o algoritmo do valentão usando uma classe `Leader` que armazena o estado necessário para gerenciar a eleição de líder, mantendo informações como uma lista de vizinhos, o id da instância atual e o id do leader atual. Essa classe também expõe interfaces que permitem iniciar uma eleição e sincronizar o líder com as demais instâncias. Essa implementação se comunica entre instâncias usando RPC com suporte da biblioteca `xmlrpc`, a mesma usado no relógio lógico.

Na inicialização de uma instância de `Leader`, recebemos o id do processo atual, uma lista de vizinhos e uma função que mapeia um id para uma instância remota de `Leader`. Também inicializamos o líder para nulo, pois ainda não houve nenhuma eleição, e a variável `running_election` para False, pois nenhuma eleição está rodando no momento. Quando um processo precisa consultar o líder atual, ele pode consultar essa propriedade diretamente.

O principal método da classe `Leader` é `start_election`. Ele é encapsulado por checagens a variável `running_election`, pois esse método pode ser demorado e não há necessidade de duas instâncias desse método rodarem simultaneamente. Esse método solicita a todos os vizinho que iniciem a eleição chamando `receive_election` percorrendo a lista de vizinhos do maior para o menor até alcançar o id atual. Ele para a procura assim que um vizinho responder que fará a eleição, pois todos os vizinhos que seriam questionados depois teriam id menor. Se não houverem vizinhos candidatos ou todos os vizinhos esiverem indisponíveis, a instância se declara líder e anuncia o resultado da eleição usando `disclose_result`.

O método `disclose_result` percorre a lista de vizinhos chamando `receive_result`, que simplesmente atualiza o líder na instância atual.

Dificuldades

Muitas das dificuldades da solução, como a comunicação entre as instâncias, já haviam sido resolvidas durante a implementação do relógio lógico, então a implementação do algoritmo de eleição de líder foi muito mais fácil. O algoritmo em si era simples o suficiente, então também não tivemos complicações, mesmo com a lógica de sincronização entre instâncias.

Ainda assim, tivemos um problema com a biblioteca `xmlrpc` em que ela lançava erros que não afetavam o funcionamento do algoritmo, mas ainda assim aconteciam. Acontece que, por padrão, essa biblioteca espera que as chamadas RPC tenham retorno, porém alguns métodos eram usados somente para sincronizar as instâncias, sem nenhum retorno para a chamada RPC, o que lançava a exceção. Corrigimos esse problema colocando um retorno `True` nos nossos métodos, mas poderíamos passar a opção `allow_none=True` para o servidor RPC para que o retorno nulo fosse aceito.

Também tivemos um problema em que o líder precisa de uma função de mapeamento de id para líder RPC, porém durante a inicialização do líder os demais servidores RPC ainda não subiram, então iniciar um cliente RPC nesse momento para criar a função de mapeamento lançava um erro. Optamos por inicializar esse parâmetro como nulo e, depois da criação dos servidores, criar os clientes RPC e criar a função de mapeamento e somente aí definí-la na classe de eleição de líder.

Testes

Para testar essa implementação, fizemos um programa simples que realiza um health check nas demais instâncias de tempos e tempos e, se necessário, vota no novo líder. A partir das mensagens de log conseguimos observar o correto funcionamento da eleição de líder:

- quando o líder atual morre, alguma das instâncias inicia a eleição e todas convergem para o mesmo resultado;
- quando ocorrem repetidas eleições, a queda do líder é detectada e a eleição é sempre chamada;
- mesmo que outros possíveis líderes tenham caído, o líder correto é sempre escolhido.

Podemos simular a indiponibilidade do líder usando um `docker-compose kill <process-hash>`. Podemos descobrir os hashs dos processos rodando `docker ps`.

Exclusão mútua

Implementação

Implementamos a exclusão mútua usando o algoritmo centralizado. Para escolher o coordenador do algoritmo, usamos a eleição de líder implementada anteriormente. Já para coordenar o próprio algoritmo implementamos duas classes, `MutualExclusionLeader` e `MutualExclusionClient`. que mantém o estado necessário para garantir a exclusão mútua. A comunicação entre as instâncias foi implementada usando a biblioteca `xmlrpc`.

A classe `MutualExclusionLeader` mantém uma fila de instâncias esperando a liberação de recursos, um boolean indicando a disponibilidade do recurso, um semáforo para controlar o acesso a fila e uma função que mapeia um id de instância para o cliente RPC da instância. Essa classe possui dois métodos principais, `receive_request` e `release_mutex`. O primeiro recebe um pedido de acesso ao recurso e, se estiver disponível, libera o acesso, senão, enfileira o processo para ser liberado futuramente. O outro método, `release_mutex`, recebe uma notificação de que o atual uso do recurso foi encerrado e, caso haja algum processo na fila, o recursos já é imediatamente repassado para este.

A classe `MutualExclusionClient` libera o recurso para um uso específico, ou seja, é possível que uma mesma instância solicite o recurso várias vezes antes que o recurso seja liberado uma única vez. Para isso, ela mantém uma lista de callbacks, para cada vez que o recurso foi liberado, um ponteiro para o cliente RPC do `MutualExclusionLeader`, o id do processo atual e um semáforo para evitar várias comunicações com o líder ao mesmo tempo. A classe expõe três métodos: `request`, `allow`, `release`. O primeiro recebe umm callback e solicita o recurso ao líder e, se esse estiver livre, executa o callback imediatamente, caso contrário, enfileira o callback para ser executado quando o recurso for liberado. O método `allow` é usado pelo líder para liberar o recurso tardiamente, chamando o primeiro callback da lista do cliente. Já o método `release` é chamado ao final do callback para informar o líder do fim do uso do recurso e liberá-lo.

Dificuldades

Tivemos um problema durante a execução do cliente pois já que ele havia sido desenhado para que cada acesso ao recurso cumprisse um objetivo específico, poderia ocorrer mais de uma comunicação com o líder ao mesmo tempo, o que causava uma exceção na biblioteca `xmlrpc`. Para lidar com esse problema, envolvemos as chamadas ao líder em um semáforo, evitando comunicações paralelas.

Isso porém trouxe outros problemas. Como o método `allow` do cliente chama um callback possivelmente lento, liberar o mutex causava um *deadlock*. Para resolver esse problema, fizemos `allow` iniciar o callback em uma thread própria e retornar mais cedo, evitando *deadlocks* na liberação do mutex.

Testes

Para testar a exclusão mútua, fizemos uma aplicação que simplesmente solicita o mutex de tempos em tempos. Ao obtê-lo, simplesmente o segura por uma porção de tempo aleatória e depois o libera.

Devido a aleatoriedade dos testes, conseguimos observar alguns casos que comprovam o funcionamento do algoritmo:

- quando o mutex está disponível, ele é entregue automaticamente a quem o soliciar;
- quando o mutex está bloqueado, ele não é entregue e a requisição é enfileirada;
- quando o mutex é liberado e a fila não está vazia, o mutex é entregue para o próximo item da fila;
- quando o mutex é liberado e a fila está vazia, o mutex fica livre e é entregue imediatamente para a próxima requisição.