MC714 - Trabalho 2

Guilherme Guidotti Brandt (235970)

1 Introdução

O projeto foi desenvolvido utilizando a linguagem Elixir (`https://elixir-lang.org/`) com deploy feito em um cluster local de Kubernetes (`https://minikube.sigs.k8s.io/docs/`) para simular uma situação de sistema distribuído.

Os algoritmos escolhidos para implementação foram os seguintes:

- Relógio lógico de Lamport (obrigatório)
- Algoritmo de Ricart–Agrawala para exclusão mútua (uma otimização do algoritmo de Lamport que vimos em classe)
- Algoritmo Paxos de consenso distribuído

Todos os algoritmos foram implementados de forma a compor uma aplicação que oferece interface HTTP, pela qual um cliente pode adicionar itens a uma lista; a lista é feita consistente entre todas as réplicas do serviço (que dividem o tráfego da API HTTP entre si através do balanceador de carga do Kubernetes) através do algoritmo Paxos, que usa o algoritmo de exclusão mútua para evitar conflitos entre múltiplas réplicas tentando conduzir unidades de consenso. O algoritmo de exclusão mútua, por sua vez, faz uso de um relógio lógico para fazer ordenação total de mensagens entre os nós.

2 Troca de mensagens

A troca de mensagens no sistema é feita usando o mecanismo nativo oferecido pela linguagem Elixir (que foi, inclusive, um dos fatores considerados na decisão de usar a linguagem).

Em Elixir, a unidade fundamental de concorrência e estado são processos. Todo código Elixir executa dentro de um processo, e cada processo é isolado dos demais e executa concorrentemente; toda comunicação entre processos se dá por meio de troca de mensagens.

Uma das grandes vantagens de Elixir (que a linguagem na verdade herda da plataforma OTP do Erlang, sobre a qual Elixir é construída) para o desenvolvimento de sistemas distribuídos é que a linguagem oferece nativamente formas de conectar diferentes instâncias de máquinas virtuais da linguagem (que podem ou não estar sendo executadas em máquinas diferentes) em um *cluster*, de

forma que cada máquina virtual (nó) no *cluster* consegue enviar mensagens a processos em qualquer outro nó.

Além das ferramentas nativas para conexão de VMs Elixir, existem bibliotecas que permitem formar *clusters* de forma automática, por exemplo através de APIs do Kubernetes, como a `libcluster` (`https://github.com/bitwalker/libcluster`), que foi usada no projeto para conectar as réplicas.

3 Algoritmos

Esta seção descreve os algoritmos e suas implementações.

3.1 Relógio lógico de Lamport

O algoritmo do relógio lógico é bastante simples: temos um número (um *timestamp* lógico) que cresce em 1 cada vez que realizamos uma ação localmente (e.g. receber uma requisição, enviar uma mensagem, etc.), e que é ajustado toda vez que recebemos uma mensagem de outro nó (cada mensagem enviada entre os nós contém um *timestamp* lógico; o relógio é ajustado para refletir o *timestamp* mais recente entre o armazenado localmente e o recebido em mensagens de outros nós).

No sistema, cria-se uma instância de relógio lógico para cada instância de trava de exclusão mútua (poderia-se utilizar uma única instância de relógio lógico para todo o sistema, mas isso poderia gerar um gargalo). Cada instância é essencialmente um processo Elixir que guarda um número (o *timestamp*), e recebe mensagens de dois tipos:

- `tick()` – Avança o relógio lógico em uma unidade (chamado quando o algoritmo utilizando o relógio lógico realiza alguma ação)
- `coalesce(t)` – Compara `t` ao *timestamp* armazenado localmente; se `t` for maior, troca o timestamp local por `t`. Ao fim dessa operação também se avança o *timestamp* em uma unidade, independente de ter sido trocado ou não.

3.2 Ricart–Agrawala

O algoritmo de Ricart–Agrawala é uma extensão e otimização do algoritmo de exclusão mútua de Lamport, que consegue realizar exclusão mútua sem a necessidade de mensagens de `ack` [2].

A ideia central do algoritmo é que um nó na rede só entra na seção crítica quando recebe permissão de todos os outros nós na rede, e cada nó só dá permissão para que outro nó acesse a seção crítica quando 1. não está na seção crítica ou não tem interesse em entrar nela ou 2. o outro nó tem prioridade no acesso (determinada usando timestamps lógicos sincronizados entre os nós, de forma a criar uma ordem total das mensagens de requisição de uso da seção crítica).

A única forma de haver um conflito (dois processos na seção crítica concorrentemente) seria se dois processos permitissem um ao outro que entrassem na seção crítica, mas isso é impossível, graças aos timestamps lógicos sincronizados (tal cenário geraria ciclo de causalidade, que é impossível).

Uma instância do algoritmo é implementada como um processo Elixir que recebe as seguintes mensagens:

- `lock(timeout)` – Pede para que o processo tente obter a trava de exclusão mútua. Uma resposta só é enviada quando 1. o processo consegue a trava, ou 2. o tempo limite de espera se esgota sem que o processo consiga a trava.

  Após receber essa mensagem, o processo envia mensagens do tipo `request_lock` para todos os outros nós da rede

- `release()` – Pede para que o processo libere a trava. Isso faz com que as mensagens que foram postergadas (ver abaixo) sejam enviadas.

- `request_lock(peer, timestamp)` – Mensagem enviada pelo nó `peer` para pedir a trava, corresponde à mensagem `REQUEST` do artigo original.

  Ao receber esta mensagem, o processo ajusta seu relógio usando o timestamp recebido, e decide se deve enviar uma resposta permitindo a trava imediatamente, ou se deve postergar a mensagem (para que seja enviada quando uma mensagem de `release` for recebida). A decisão é feita da seguinte forma:

  - Se o processo está, atualmente, em posse da trava (i.e. dentro da seção crítica), posterga a mensagem;
  - Se o processo tem interesse na trava (i.e. recebeu uma mensagem de `lock`, mas ainda não recebeu todas as mensagens de aceite), ele compara o timestamp recebido com o timestamp em que pediu a trava (que é armazenado quando se recebe uma mensagem de `lock`); caso seu timestamp seja mais recente, ele responde imediatamente; caso seja menos recente, ele posterga a mensagem; caso os timestamps sejam iguais, o desempate é feito com o nome dos nós (que são únicos).

- `allow_lock(peer, timestamp)` – Mensagem enviada pelo nó `peer` permitindo acesso ao nó atual que acesse a trava, corresponde à mensagem `REPLY` do artigo original.

  Ao receber essa mensagem, o processo remove o nó `peer` da lista de espera (que é criada contendo todos os nós da rede no momento de recebimento da mensagem de `lock`). Se a lista de espera ficar vazia, o processo entra na seção crítica (respondendo a mensagem de `lock`).

3.3 Paxos

O algoritmo Paxos resolve o problema do consenso distribuído [1]. O algoritmo pode ser divido em duas "partes": uma unitária (que Lamport chama

de “sínodo”), que garante que os nós de uma rede concordem em um único valor (que, uma vez decidido, não se altera), e uma sequencial, que faz uso da unitária, e permite que os nós concordem com uma sequência de valores, que podem, por exemplo, ser usados como entrada de uma máquina de estados arbitrária (que, graças ao algoritmo, seria sequencialmente consistente entre todos os participantes).

3.3.1 Sínodo

A parte unitária do algoritmo funciona através de “eleições”. Qualquer nó na rede pode pedir o início de uma eleição, enviando mensagens para o conjunto de nós ”aceitadores”(mais sobre isso na seção de máquina de estado). Tais nós, então, respondem informando se participarão na eleição (e se vão, se já votaram em algum outro valor em uma eleição anterior), ou se já participaram de outra mais recente e não podem participar desta.

Se alguma maioria dos nós aceitadores concordar em participar da eleição, o nó que a iniciou decide o valor que será escolhido da seguinte forma:

- Se nenhum dos nós que concordou em participar da eleição votou em alguma eleição anterior, pode escolher qualquer valor (no caso de uma sequência de valores, aqui seria escolhido o valor que se deseja incluir na sequência, por exemplo).
- Caso contrário, escolhe-se o valor em que votou o nó que votou na eleição mais recente entre os que aceitaram a eleição (o caso de empate não é um problema, porque todo voto em uma mesma eleição é para o mesmo valor).

O nó que inciou a eleição então informa aos nós que concordaram em participar do valor escolhido, e cada um decide votar ou não. Um nó vota em uma eleição se e somente se ele não concordou em participar de nenhuma eleição mais recente (as eleições são identificadas por um número de sequência e um nó, e comparadas lexicograficamente; uma eleição é mais recente que outra se tem um número de sequência maior, ou um número de sequência igual e um identificador de nó maior).

Se todos os nós que concordaram com a eleição votarem, o algoritmo chegou a um consenso, e o nó que iniciou a eleição informa todos os demais qual o valor escolhido.

Sempre, antes de enviar uma mensagem para outro nó, cada nó salva seu estado em armazenamento persistente (no caso, um volume persistente do Kubernetes, que fica relacionado a cada pod no StatefulSet de forma que se o pod falha, quando reinicia ele tem acesso ao mesmo volume). Dessa forma, garante-se que a consistência é preservada mesmo em caso de falha dos pods.

Esse algoritmo garante que uma vez que uma eleição é bem-sucedida (i.e. um valor é decidido e todos votam), todas as eleições subsequentes decidem o mesmo valor (teorema 1 em [1]).

Uma instância do algoritmo é implementada em Elixir como um processo que recebe as seguintes mensagens:

- `propose(value)` – Propõe um valor para ser decidido. Esse valor é usado caso uma eleição aconteça em que nenhum dos participantes votou em alguma eleição anterior.

  O recebimento dessa mensagem faz com que o nó atual comece uma eleição, enviando mensagens de `request_ballot` para todos os demais nós da rede.

- `request_ballot(peer, ballot)` – Mensagem enviada pelo nó `peer` pedindo para iniciar a eleição de número `ballot`. O nó receptor responde com uma mensagem `ballot_accept` ou `ballot_reject`, seguindo a lógica do algoritmo para participar ou não de uma eleição.

- `ballot_accept(peer, ballot, max_ballot, max_ballot_value)` – Mensagem enviada pelo nó `peer` concordando em participar na eleição `ballot`. A mensagem também contém `max_ballot` (a eleição mais recente na qual o remetente votou, ou -1 caso não tenha votado em nenhuma) e `max_ballot_value` (o valor em que votou na eleição, ou `nil` caso não tenha votado em nenhuma).

  Ao receber essa mensagem, o nó que iniciou a eleição armazena a informação recebida, e verifica se pode começar a eleição (i.e. uma maioria dos nós que atuam como aceitadores concordou em participar dela); caso positivo, envia mensagens `begin_ballot` para todos os participantes.

- `ballot_reject(peer, ballot, max_ballot)` – Mensagem enviada pelo nó `peer` rejeitando participar da eleição `ballot`. A mensagem também contém `max_ballot`, a eleição mais recente da qual o remetente participou.

- `begin_ballot(peer, ballot, value)` – Mensagem enviada pelo nó `peer` começando a eleição `ballot` com valor `value`. Ao receber essa mensagem, o nó atual verifica se deve votar na eleição, seguindo a lógica do algoritmo, e vota caso positivo.

- `vote(peer, ballot)` – Mensagem enviada pelo nó `peer` votando na eleição `ballot` com valor `value`.

  Ao receber essa mensagem, o nó que iniciou a eleição armazena a informação recebida e verifica se todos votaram na eleição. Caso positivo, envia mensagens de sucesso a todos os nós na rede.

- `success(value)` – Indica que o consenso foi atingido com o valor `value`. Ao receber essa mensagem, um nó entra em estado de sucesso e armazena o valor recebido (de forma que pode informar a outros nós o valor decidido, e não precisa começar novas eleições).

3.3.2 Máquina de estados

A máquina de estados é implementada com uma sequência de instâncias unitárias do algoritmo Sínodo. Cada nó da rede mantém uma sequência de processos implementando o algoritmo unitário, e sempre que deseja fazer alguma operação

de escrita na máquina de estados, cria novas instâncias de consenso até que em alguma se decida usar a operação proposta.

No artigo original, Lamport sugere que seja eleito um líder para conduzir cada instância de consenso, bem como um líder para propor operações na máquina de estados; isso é uma otimização que evita que várias réplicas tentem criar eleições ao mesmo tempo em uma mesma instância de consenso (entre outras vantagens). No projeto, utilizou-se o algoritmo de exclusão mútua para atingir (essencialmente) o mesmo efeito: antes de propor um valor em uma instância de consenso, o nó pede uma trava de exclusão mútua (identificada pelo número de sequência da instância, i.e. sua posição na lista de operações), de forma que dois nós nunca estarão criando eleições para a mesma instância de consenso ao mesmo tempo.

Outro detalhe importante da implementação da máquina de estados é que ela inclui também o conjunto dos nós que atuam como "aceitadores"nas instâncias de consenso: para passar a fazer parte do conjunto, cada nó precisa propor uma operação que o adiciona no conjunto (idem para remoção). Essa é uma abordagem sugerida no artigo original.

Finalmente, é possível realizar operações de leitura consistentes através do mecanismo de propostas de operações na máquina de estados: basta propor uma operação "noop", que não tem efeito sobre a máquina de estados; quando a operação é realizada, tem-se certeza que ela foi a operação mais recente em todas as réplicas, então basta ler o estado atual da máquina de estados para obter o estado mais recente.

3.4 Testes

Como a aplicação e os algoritmos são bastante integrados, o principal teste feito foi um "teste de carga", onde um volume elevado de dados era inserido na lista mantida pelo serviço (e, para aumentar o efeito de caos, ao mesmo tempo que novas réplicas eram criadas).

O resultado foi relativamente satisfatório: o consenso não se desfez, e a aplicação não entrou em deadlock (as duas principais condições de correção dos algoritmos usados!). Algumas das requisições de inserção de dados falharam, o que provavelmente se deu por conta da falta de healthchecks adequados nas réplicas sendo inicializadas (que provavelmente não estavam em condições de responder requests imediatamente), mas isso poderia ser mitigado usando medidas de resiliência simples (em particular, retries).

Referências

[1] L. Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. 2019.

[2] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, jan 1981.