Engenharia de Telecomunicações — Prof. Emerson Ribeiro de Mello

STD29006 – Sistemas Distribuídos — http://docente.ifsc.edu.br/mello/std

Projeto prático 2: Replicação primário e secundários

18/08/2021

1 Descrição do problema

O método de replicação primário e secundários é usado em muitos softwares, como por exemplo, banco de dados relacionais. Neste método, toda operação de atualização do estado do sistema sempre é encaminhada para o processo primário e compete a esse encaminhar o *log* com as alterações para as réplicas, como ilustrado pela Figura 1. A replicação pode ser:

- **Síncrona** – cliente não receberá resposta até que o primário consiga aplicar as alterações em todas as réplicas; ou
- **Assíncrona** – cliente receberá resposta imediata do primário e esse encaminhará também de forma assíncrona as alterações para os secundários (réplicas).

No modo síncrono é necessário fazer uso de duas mensagens: *update* – enviada pelo primário; e *acknowledge receipt* – enviada pelos secundários. Já no modo assíncrono somente a mensagem *update* se faz necessária. O modo síncrono provê garantias que a escrita foi realizada em todas as réplicas antes de retornar uma resposta para o cliente, porém ao custo de deixar o cliente aguardando até que isso aconteça.

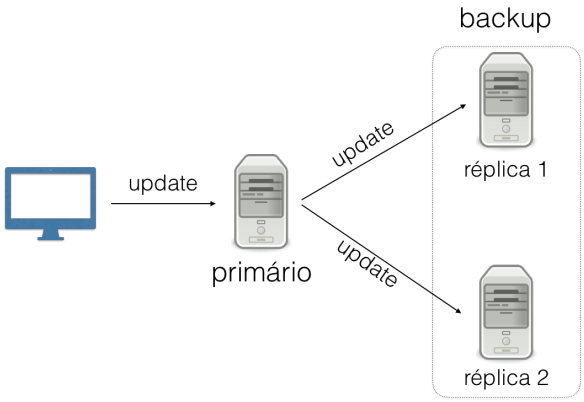

Figura 1: Replicação primário e secundários

O protocolo de *commit* em duas fases (*Two-phase Commit* – 2PC) é usado pelo MySQL Cluster, um banco de dados relacional distribuído (não confundir com o MySQL convencional), para permitir a replicação síncrona entre primário e secundários. A Figura 2 ilustra a troca de mensagens do protocolo 2PC em um cenário com um coordenador (primário) e mais duas réplicas.

Na primeira fase o coordenador envia uma ação de atualização (*update*) a todas as réplicas e questiona se essa atualização pode ou não ser persistida. Cada réplica deve então: (1) armazenar essa ação em uma memória não volátil (*write-ahead log*) como um registro de intenções, ou seja, de ações que pretende concretizar, mas que ainda não o fez; (2) enviar uma resposta ao coordenador, sendo *yes* para indicar que concorda em persistir permanentemente a ação ou *no* para indicar não concorda em persistir permanentemente.

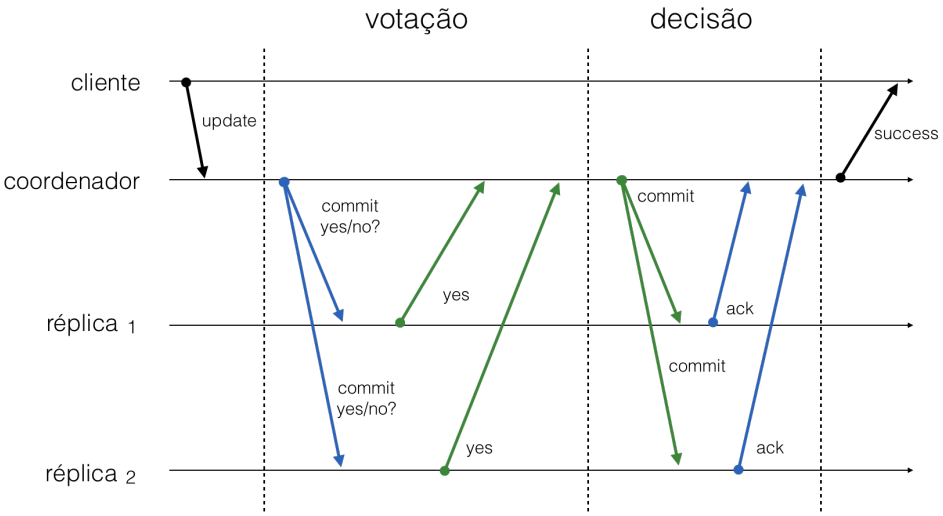

Figura 2: Trocas de mensagens entre coordenador e réplicas no protocolo 2PC

Na segunda fase do protocolo o coordenador, após coletar as respostas de todas as réplicas, deve tomar uma decisão. Se todas as respostas forem *yes*, então o coordenador envia uma mensagem *commit* para todas as réplicas. Porém, se pelo menos uma das respostas for *no*, então o coordenador enviar uma mensagem *abort* para todas as réplicas. Uma réplica ao receber uma mensagem *commit* deve retirar a ação de sua memória temporária e persistir de forma permanente. Contudo, se receber uma mensagem *rollback*, então só deve retirar a respectiva ação de sua memória temporária. Em ambos os casos a réplica precisa enviar uma mensagem *ack* para o coordenador. Na Figura 2 é ilustrado um exemplo onde a ação de atualização, enviada pelo cliente, foi de fato persistida em todas as réplicas.

2 Solução a ser desenvolvida

Desenvolva uma aplicação que forneça uma API RESTful que permita realizar uma simulação do protocolo de *commit* em duas fases. Para essa simulação assuma que os processos estarão 100% do tempo disponíveis e nunca irão falhar. A aplicação a ser desenvolvida poderá assumir o papel de coordenador ou de réplica, cabendo o usuário indicar qual o papel de cada instância da aplicação. Assuma que o cenário para experimentos sempre terá 1 processo como coordenador e 2 processos como réplicas.

A aplicação a ser desenvolvida deverá persistir em disco (ou em memória volátil) informações sobre contas bancárias. Para cada conta é necessário guardar seu número único e o saldo. Com o intuito de simplificar, sempre que o processo for iniciado as seguintes informações deverão ser carregadas:

Conta	Saldo
1234	100,00
4345	50,00
5678	250,00

A API a ser desenvolvida deverá permitir as seguintes funcionalidades:

- **Carregar lista de réplicas** – Deverá receber um documento JSON com o *endpoint* de cada réplica, além de seu identificador único. Exemplo:

```
{
  "replicas" : [
    {
    "id" : "replica 1",
    "endpoint" : "http://192.168.0.1/pp02"
    },
    {
    "id" : "replica 2",
    "endpoint" : "http://192.168.0.15/pp02"
    }
  ]
}
```

  - Ao invocar essa funcionalidade, o processo em questão será definido como coordenador. Sendo assim, você terá em execução 3 processos que implementam a mesma API RESTful, mas de somente um deles você poderá invocar essa funcionalidade. Isso é uma limitação deste exemplo que visa simular o 2PC e na prática não teríamos tal funcionalidade.

- **Apagar lista de réplicas** – Deverá apagar a lista de réplicas, caso essa tenha sido carregada, por exemplo, pela funcionalidade do item anterior. Deve-se também indicar que o processo não é mais coordenador;

- **Obter lista de réplicas** – Deverá retornar um documento JSON contendo a lista de réplicas no mesmo formato do JSON apresentado acima;

- **Obter lista de contas** – Deverá retornar um documento JSON com as contas e seus respectivos saldos. Exemplo:

```
{
  "contas" : [
    {
    "numero" : 1234,
    "saldo" : 100.00
    },
    {
    "numero" : 4345,
    "saldo" : 50.00
    }
  ]
}
```

- **Enviar ação** – Deverá receber um documento JSON com a ação que deverá ser enviada para todas as réplicas. A ação poderá ser um saque (débito) ou depósito (crédito) de uma quantia em uma determinada conta. Toda ação deverá ter um identificador único. Exemplo:

```
{
  "id" : "19148f6d-1318-4887-b2b6-215bfc8ac35f",
  "operacao" : "debito",
  "conta" : 1234,
  "valor" : 10.00
}
```

  - **Se o processo for coordenador**, então esse deverá persistir os dados na área de armazenamento temporário (*write-ahead log*) e invocar essa funcionalidade "Enviar ação" das réplicas e depois retornar o código `HTTP 201 Created` para representar a mensagem *success* do 2PC ou `HTTP 403 Forbidden` para representar a mensagem *fail* do 2PC. Em caso de sucesso, deve-se persistir os dados da ação de forma permanente.
  - **Se o processo for réplica**, então esse deverá persistir os dados na área de armazenamento temporário (*write-ahead log*) e deverá retornar o código `HTTP 200 OK` para representar a mensagem *yes* do 2PC ou o código `HTTP 403 Forbidden` para representar a mensagem *no* do 2PC. Faça um sorteio com `Random` para determinar se a resposta será *yes* ou *no*. **A probabilidade de retornar *yes* deverá ser de 70%**.
  - A área de armazenamento temporário poderá ser persistida em memória ou em disco.

- **Enviar decisão** – Poderá ser consumido somente com o verbo `HTTP PUT` para representar a mensagem *commit* do 2PC ou com o verbo `HTTP DELETE` para representar a mensagem *rollback* do 2PC. Deve-se aqui fornecer um documento JSON contendo o identificador único da ação. Exemplo:

```
{
    "id" : "19148f6d-1318-4887-b2b6-215bfc8ac35f"
}
```

  - **Se o processo for coordenador**, então deverá retornar o código `HTTP 400 Bad Request` independente do tipo do verbo (`PUT` ou `DELETE`).
  - **Se o processo for réplica**, então deverá persistir os dados da ação de forma permanente (se receber `PUT`) ou remover a ação da área temporária (se receber `DELETE`). Para ambos os casos deve-se retornar o código `HTTP 200 OK`. Se o identificador da ação não existir, então deve-se retornar o código `HTTP 404 Not Found`.

- **Obter histórico de ações processadas** – deverão retornar um documento JSON com a lista de ações que foram processadas. Esse documento deverá conter o identificador único de cada ação e a situação da mesma: *success* se ela foi processada ou fail, caso contrário.

```
{
  "acoes" : [
    {
    "id" : "19148f6d-1318-4887-b2b6-215bfc8ac35f",
    "status" : "success"
    },
    {
    "id" : "0fcf8b5f-622b-4923-81c4-43b1753e403f",
    "status" : "fail"
    }
  ]
}
```

- **Carregar semente** – Deverá receber um número inteiro em um documento JSON que deverá ser usado como semente do gerador de números pseudo aleatórios, que é usado quando o processo for uma réplica durante a fase de votação. Exemplo:

```
{
  "seed" : 123456
}
```

3 Entregas

- **Código fonte da aplicação – 8 pontos**
  - Código da aplicação funcionando corretamente e com todas as funcionalidades. Pode ser feito em Java ou em Python;
  - Disponibilização da aplicação em uma composição com Docker Compose;
  - Garanta que seja possível compilar e executar o código após o clone. Caso não seja possível, então a nota será 0 nesse item
- **Documentação da API – 1 ponto**
  - Documento na raiz do repositório com o nome `apiary.apib` e este deve conter a documentação da API REST de acordo com a especificação API Blueprint[1]
  - Veja um exemplo em https://github.com/std29006/laboratorio-REST-java-gradle
  - Em https://apiblueprint.org/documentation/examples/ tem uma documentação sobre a API Blueprint
  - **Sugestão:** Para escrita da documentação faça uso do editor *Visual Studio Code* com as extensões *API Blueprint Viewer* e *API Elements extension*
- **Relatório – 1 ponto**
  - Arquivo `Readme.md` na raiz do repositório onde deve-se indicar como compilar e executar o experimento, apresentando alguns exemplos para fazer saques e depósitos;
  - Deve-se ainda indicar quais funcionalidades foram implementadas e quais não foram
- Data para entrega: **05/09/2021**
  - Somente via Github Classroom.

[1] https://apiblueprint.org/