ForkExec – Relatório do grupo A45

https://github.com/tecnico-distsys/A45-ForkExec

Gabriel Figueira, 86426 Lívio Costa, 86461 Rafael Andrade, 86503

Modelo de faltas

Segundo o enunciado:

- Os gestores de réplica podem falhar silenciosamente mas não arbitrariamente, i.e., não há falhas bizantinas;
- No máximo, existe uma minoria de gestores de réplica em falha em simultâneo;
- O sistema é assíncrono e a comunicação pode omitir mensagens (apesar do projeto usar HTTP como transporte, deve assumir-se que outros protocolos de menor fiabilidade podem ser usados)

Na nossa implementação, as falhas silenciosas toleradas são, por exemplo:

- Não estar inscrito no *UDDI*
- Estar inscrito no *UDDI*, mas demorar tempo indeterminado a responder

Solução de tolerância de faltas

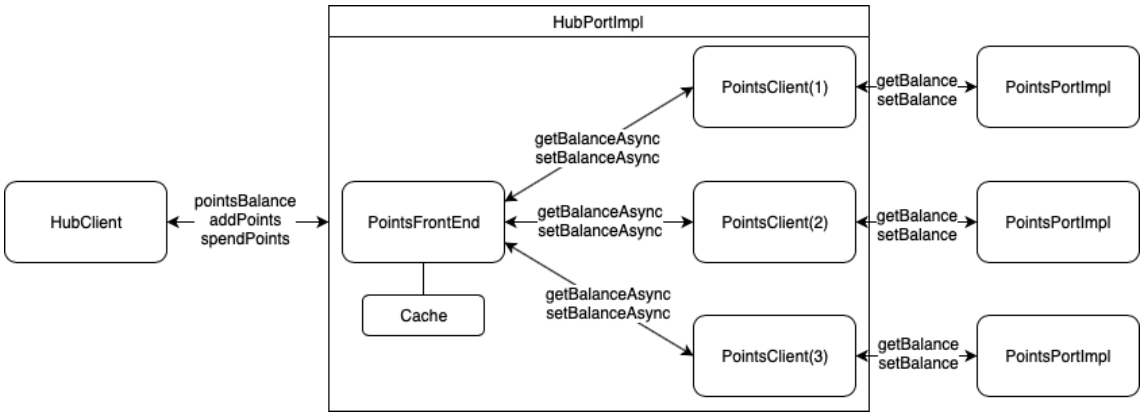

Figure 1: Organização da nossa solução

A nossa solução passa por utilizar o algoritmo *Quorum Consensus*, onde:

- Cada gestor de réplica é um servidor de pontos com a informação de todos os clientes do *Hub*.
- O *Front End* garante que só há **um** acesso (leitura ou escrita) para o valor de pontos para cada cliente, e portanto os servidores de pontos não terão mecanismos para garantir que não há acessos concorrentes do mesmo valor (i.e., a consistência sequencial do protocolo é assegurado somente no *Front End*).

Em termos concretos, a classe `PointsFrontEnd` dentro do módulo `pts-ws-cli` é que implementa os métodos `read` e `write` necessários para o *Quorum Consensus*, encapsulando assim a utilização deste protocolo; e para ainda tornar esta mudança menos drástica em termos de alteração de código, o `PointsFrontEnd` implementa a antiga interface do `PointsClient`[1].

[1]Na implementação, foi necessário alterar as exceções e alguns testes, uma vez que não podemos assumir que um utilizador não existe só porque não se encontra registado nas múltiplas réplicas.

Por exemplo, o spendPoints agora é implementado assim (pseudo-código):

```
def spendPoints(user, pointsToSpend):

    lock()
    taggedBalance = read(user)

    if balance.points < pointsToSpend:
        throw NotEnoughPointsException

    taggedBalance.points -= pointsToSpend
    taggedBalance.tag += 1

    write(user, taggedBalance)
    unlock()
```

Este protocolo garante consistência sequencial.

Optimizações & Simplificações

- As mensagens trocadas entre o *Front End* e os gestores de réplica contêm o número de pontos e um número de sequência. Não existe campo para identificar o *Front End*, porque neste caso o *Front End* é único.
- O *Front End* pode fazer vários pedidos assíncronos a um mesmo servidor de pontos, mas os ditos pedidos são acerca de clientes diferentes. O servidor de pontos pode tratar destes pedidos de forma paralela.
- Se forem acerca do mesmo cliente, o *Front End* possui um mecanismo de *locks* que garante a consistência e permite a existência de múltiplos leitures ou um escritor (implementado utilizando `ReadWriteLock`).
- O *Front End* tem um mecanismo de *cache*, que minimiza os acessos necessários aos gestores de réplicas. Sendo assim, estes (os gestores de réplica) funcionam como *fallback*, caso o valor pretendido não esteja na dita *cache*. Caso houvesse a possibilidade de haver 2 *Front Ends* a realizarem operações ao mesmo tempo, seria necessário implementar um mecanismo de invalidação de *cache*.
- Caso o *Front End* detete que só consegue contactar uma minoria de gestores de réplica (o que não é tolerável de acordo com o nosso modelo de faltas), é mandada uma `RuntimeException` e a operação não é realizada.

O método de chamadas assíncronas utilizado foi *polling*, por uma questão de simplicidade na implementação. Tendo mais tempo, seria preferível utilizar uma solução de *callback*, porque permitiria evitar *busy waiting* da *thread* principal, evitando assim desperdício em termos de trabalho realizado.

Protocolo

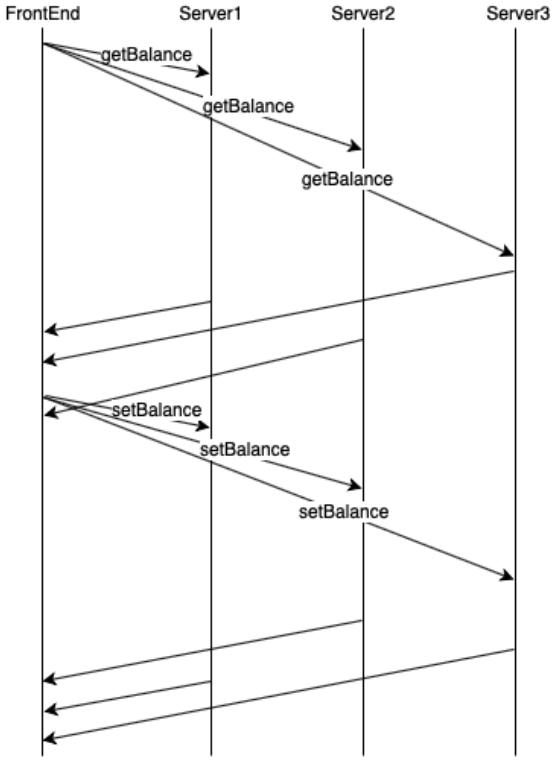

Figure 2: Troca de mensagens durante um `spendPoints`

Remoção de pontos:

- o `FrontEnd` envia pedidos de leitura aos `N` servidores de pontos
- cada servidor de pontos responde com os números de pontos e versão armazenados para esse utilizador
- o `FrontEnd` espera por `(N/2)+1` respostas, e vai guardando o maior número de versão observado e respetivo valor.
- depois de ter as respostas, o `FrontEnd` verifica que o balance é suficiente, senão lança uma exceção `NotEnoughPointsException`.
- se puder, subtrai os pontos, incrementa por 1 a versão e manda `N` pedidos de escrita
- Após ter `(N/2)+1` confirmações, retorna.

Os pedidos `getBalance` podem ser evitados se o `FrontEnd` tiver feito uma leitura recente, tendo portanto o valor dessa leitura em _cach