INSTITUTO SUPERIOR TÉCNICO

Sistemas Distribuídos

3º Ano, 2º Semestre 2018/2019

Relatório - Segunda Parte

Autores
86411 - Filipe Marques
86456 - Jorge Martins
86492 - Paulo Dias

Docente
Tomás Grelha da Cunha

A41-ForkExec

May 2, 2019

Contents

1 Definição do Modelo de Faltas

Assume-se que:

- 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
- Existem **N** gestores de réplicas e **N** é constante e igual a 3
- Os gestores de réplicas podem falhar silenciosamente mas não arbitrariamente
- No máximo, existe uma minoria de gestores de réplica em falha em simultâneo

2 Solução de Tolerância a Faltas

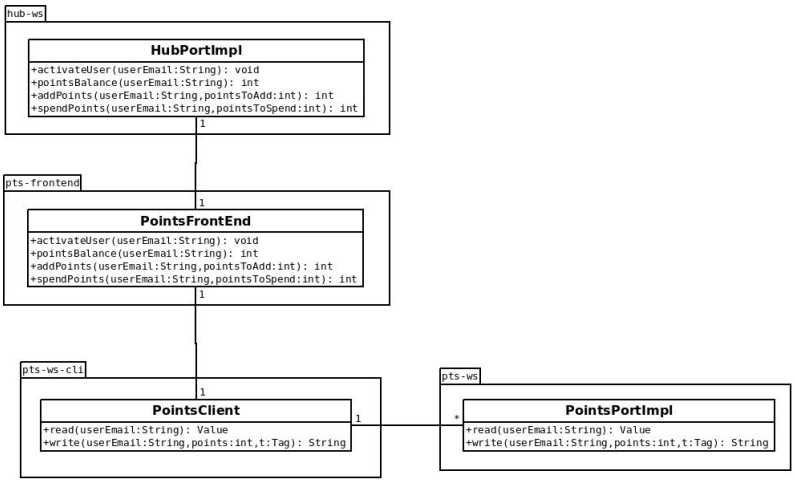

Figure 1: Simplificação do domínio da solução implementada

3 Descrição e breve explicação da solução

Para manter a interface para o *Hub* a *FrondEnd* do *points* continua a implementar as funções necessárias para retornar os valores para o *Hub*, no entanto esta classe apenas resolve a procura das replicas e reencaminha os pedidos para um *PointsClient*.

No *PointsClient* foi implementado o algoritmo *QC* que será descrito com mais detalhe no final do relatório.

4 Descrição de otimizações/simplificações

O algoritmo *Quorum Consensus* suporta vários clientes. E sempre que se pretende atualizar as replicas é necessário primeiro obter a *tag* máxima de entre todas as replicas. No entanto como foi explicito no enunciado podemos assumir que apenas existe um cliente.

Desta forma decidimos implementar uma cache no *PointsClient* que guardava as informações de cada utilizador (*tag* e *points*), com o objetivo de deixar de fazer *reads* antes de *writes*, evitando-se o tempo de consultar as replicas, sendo apenas necessário ir consultar a cache. Assim as replicas continuam com a informação dos utilizadores correta, e a funcionar normalmente. Esta otimização funciona pois apenas existe um cliente e toda a informação passa por este.

Por exemplo se existissem dois clientes esta otimização não funcionaria pois o cliente 1 poderia ter na *cache* o valor X enquanto o cliente 2 escrevia nas replicas Y, e quando fosse pedido o valor ao cliente 1 este não consultaria as replicas apenas a cache e devolveria e devolveria X, tornando-se um algoritmo inconsistente.

5 Detalhe do protocolo (troca de mensagens)

5.1 Funções no Gestor de Réplica

5.1.1 *read(String userEmail)*

- Ao receber *read(userEmail)*:
  1. Vai buscar a *tag* e os *pontos* associados ao *userEmail*
     1.1. Se *userEmail* não existe no sistema, então adiciona.
  2. responde com $Value =< pontos, tag >$ associado ao utilizador, em que: $tag =< seq, cid >$

5.1.2 *write(String userEmail, int points, Tag t)*

- Ao receber *write(userEmail,points, t)*:
  1. Vai buscar a *tag* associada ao *userEmail*
     1.1. Se *userEmail* não existe no sistema, então adiciona.
  2. Se *t.getSeq() > tag.getSeq()*:
     2.1. atualiza os *pontos* do utilizador com *points*
     2.2. atualiza a *tag* do utilizador com *t*
     2.3. responde *ack*
  3. Senão responde *nack*

5.2 Funções no *Points Client*

5.2.1 Simplificação

Para simplificar e reduzir a quantidade de texto repetido vamos definir o comportamento das seguintes funções:

getMaxValue(String userEmail)

- Para cada replica:
  - faz uma chamada assíncrona (*readAsync(userEmail)*) para ir buscar o *maxValue*
- Enquanto o *numRespostas* for menor que *Q*.
  - Para cada chamada
    - Se já chegou, guarda o *Value*
    - *numRespostas++*
- Determina e retorna *maxValue*

setMaxValue(String userEmail, Value value)

- Para cada replica:
  - executa *writeAsync(userEmail, value.getVal(), value.getTag())*
- Enquanto *numRespostas* for menor que *Q*
  - Para cada chamada
    - Se foi recebido *ack*, *numRespostas++*
- Fim

5.2.2 *activateUser(String userEmail)*

- maxValue = getMaxValue(userEmail)
- Se a sequencia associada ao *maxValue* é maior que 0 então é porque o utilizador existe e:
  - *throw EmailAlreadyExists*
- Senão:
  - *setMaxValue(userEmail, newValue(maxValue.getVal(), maxValue.getTag()))*
  - *addUserToCache(userEmail, maxValue)*

5.2.3 *pointsBalance(String userEmail)*

- vai buscar o valor associado ao userEmail à cache
  - senão existe:
    - *throw InvalidEmail*
- *return valueFromCache*

5.2.4 *addPoints(String userEmail, int pointsToAdd)*

- *value* = valor associado a *userEmail* na cache
- *points = value.getVal() + pointsToAdd*
- *setMaxValue(userEmail, newValue(points, value.getTag()))*
- atualizar a cache

5.2.5 *spendPoints(String userEmail, int pointsToSpend)*

- *value* = valor associado a *userEmail* na cache
- *points = value.getVal() - pointsToAdd*
- *setMaxValue(userEmail, newValue(points, value.getTag()))*
- atualizar a cache