# Sistemas Distribuídos

2º Semestre - 2018/2019

## ForkExec: Relatório

Grupo T26 - https://github.com/tecnico-distsys/T26-ForkExec


| | | |
|---|---|---|
| João Miguel P. Campos | 75785 |  |
| Gonçalo Santos | 77915 |  |
| Alexandre Caldeira | 85254 |  |

## Modelo de Falhas

Num sistema distribuído, tal como o que foi trabalhado neste projecto, enquanto *developers*, temos que assumir algum tipo de falhas, como por exemplo, o caso onde os servidores estejam offline, ou haja algum problema na conexão entre os servidores. Assim, é necessário que haja tolerância a essas falhas, ou seja, se um servidor falhar, temos que garantir que o sistema continua a operar da forma prevista.

O tipo de falhas que iremos tolerar são silenciosas, que são falhas onde, ou o sistema está completamente operacional, ou deixa de estar disponível, tornando-se silencioso.

Para coordenar a escrita e leitura nas N réplicas do servidor vamos usar o protocolo Quorum Consensus onde cada réplica vai receber um objecto que contém uma tag com um Sequence ID que corresponde ao número de sequência da escrita que deu origem a essa versão.

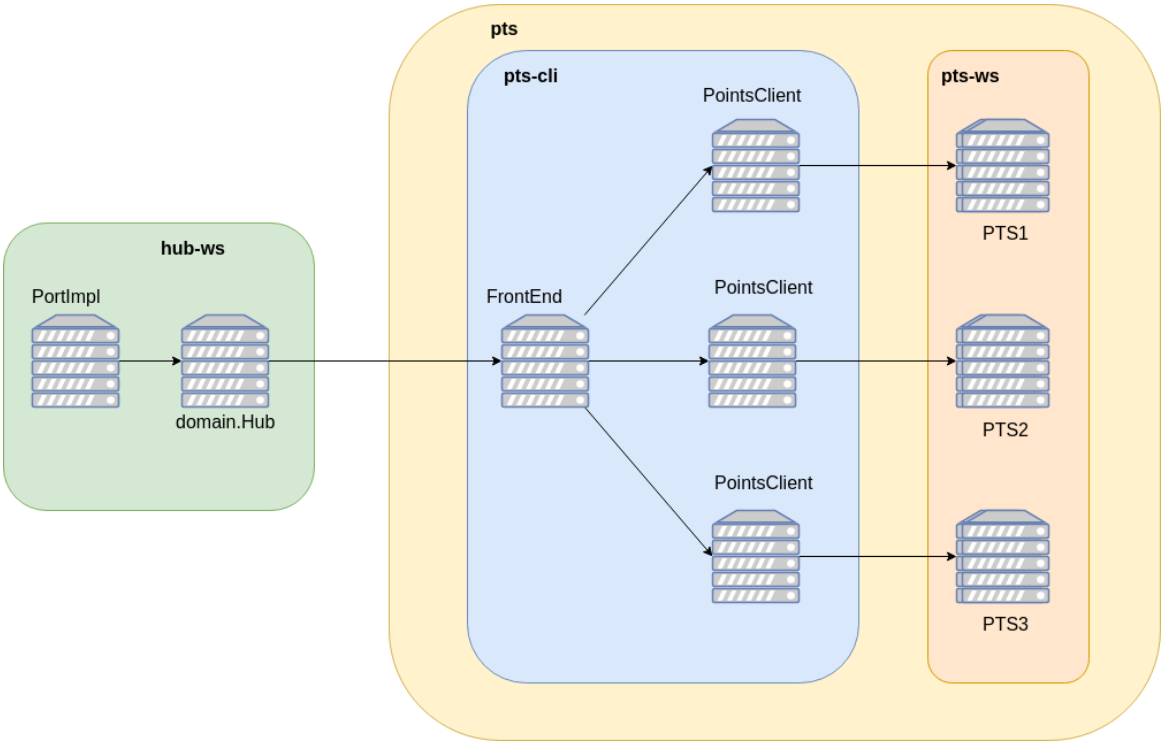
O protocolo *Quorum Consensus* é usado, pois garante que estando o *Client* a ler uma variável x e não estando a decorrer nenhuma escrita concorrente da mesma variável, o valor valor devolvido vai corresponder ao valor mais recente possível, tal é garantido devido ao facto da operação de leitura receber os valores de um quórum de *x*, enquanto a escrita anterior atualizou um quórum de *x*, e como a escrita
mais recente tem um valor de *Tag* superior a leitura vai escolher esse valor.
É importante referir que cada quórum de escrita *x* tem pelo menos uma réplica em comum com cada quórum de leitura *x* ou quórum de escrita de *x*.

As vantagens do Quorum Consensus incluem o facto de tolerar falhas silenciosas em sistemas assíncronos e que se uma réplica falhar temporariamente, e depois recupere, estará imediatamente pronta para participar e ficará naturalmente atualizada quando receber o próximo pedido de escrita.

As desvantagens deste protocolo incluem o facto de necessitar que muitas réplicas para tolerar um número curto de falhas , dado que geralmente são precisas *2*f+1* réplicas para tolerar *f* falhas de réplicas.

Outra desvantagem inclui o facto de leituras implicarem respostas de múltiplas réplicas e dado que em muitos sistemas as leitura são predominantes, então o ideal seria permitir que a leitura retornasse após a resposta de uma réplica apenas.

**Descrição do diagrama**




No diagrama, existem 3 grandes componentes: ***hub-ws***, ***pts-cli*** e ***pts-ws***.

O ***hub-ws*** é o cliente geral do serviço PTS. É dele que vêm todos os pedidos feitos ao servidor pts, e, da forma como pensamos a solução, ele não tem conhecimento da replicação do servidor de pontos.

O ***pts-cli*** é chamado pelo ***hub-ws*** sempre que houver necessidade de alterar os pontos de um utilizador ou criar um utilizador novo. O ***pts-cli*** tem uma interface, o *FrontEnd*, que disponibiliza os métodos que estão acessíveis ao seu cliente.

Também é no *FrontEnd* que é implementado *Quorum Consensus*, ou seja, o *FrontEnd* vai comunicar com o cliente de cada servidor PTS, o *PointsClient,* através de *reads* e *writes*, e vai ficar à espera, de uma forma assíncrona[1], que o PointsClient retorne o resultado que o servidor PTS enviou. Ao receber a resposta de 2 dos *PointsClients*, o *FrontEnd* encontra-se disponível para enviar uma resposta ao ***hub-ws***.

O ***pts-ws*** é a parte do serviço ***pts*** que implementa toda a lógica de criação de utilizadores, adicionar e retirar pontos a um dado utilizador e também verificar o saldo dos utilizadores.

[1] De forma não bloqueante.

## Optimizações

No nosso projecto, estamos a utilizar algumas otimizações, listadas em baixo.

1. Como apenas há um cliente, ou seja, um hub, a *tag* de cada *User* apenas tem um *sequence_number* (*seq*), evitando assim a existência de um client_id (cid);
2. Apesar de nos ser dito que devemos assumir que o número de réplicas é 3, o projeto está preparado para tolerar mais réplicas, desde que se altere o valor do atributo "numReplicas" que está no ficheiro "project.properties" para o número de réplicas que vão existir. Tendo em conta que não há mais escritas do que leituras porque para escrever eu preciso sempre de ler, achámos que, em casos de haver, por exemplo, 7 réplicas, serem apenas necessárias 3 respostas a pedidos de leitura e serem necessárias 5 respostas a pedidos de escrita. Desta forma permite-se que leituras terminem mais rapidamente apesar de as escritas poderem demorar um pouco mais, mas como nunca há mais escritas que leituras, isto resulta num sistema mais eficiente.
3. No método pointsBalance(String userEmail), após obter o resultado da consulta do saldo da conta do utilizador em questão, existe uma fase de Writeback que vai funcionar como se fosse um update em todas as réplicas.

### Detalhes do Protocolo

A activação de uma conta, numa situação sem faltas, ocorre da seguinte forma:

- É feita uma activação da conta num servidor, e depois, esta é propagada para todas as réplicas de pontos.

O acréscimo e decréscimo de pontos numa conta de um utilizador, ocorre da seguinte forma:

- É feita uma leitura da conta, e caso não exista, é criada. Depois de ser reunido um consenso sobre qual é a tag mais avançada, essa tag é incrementada, e essa mudança é escrita e propagada entre todas as réplicas.

A operação para consultar o saldo de pontos do utilizador, ocorre da seguinte forma:

- É feita uma leitura a todas as réplicas, e a versão com a tag mais elevada, após reunido um consenso, é retornada ao cliente. Após conhecer a versão mais atualizada, esta versão é propagada para as restantes réplicas, para que também elas fiquem com a versão mais recente.