Instituto Superior Técnico, Lisboa
LEIC 2018/2019
Sistemas Distribuídos

Projeto ForkExec – Parte 2
Relatório
Grupo A40

65284	86448	87658
Rui Alves	João Coelho	Francisco Santos

(A última versão do código deve ser obtida através do Fénix e não do Git)

Definição do modelo de faltas

Na segunda parte deste projeto, existem agora 3 réplicas dos servidores de pontos (*pts*), responsáveis por guardar e permitir consultar o saldo dos utilizadores da plataforma, para além de terem o objetivo de preservar essa informação caso um dos servidores falhe (redundância). Usando a implementação desse servidor submetida para a primeira parte deste projeto, e não considerando faltas bizantinas, vamos apenas considerar as seguintes faltas silenciosas (que vão ser recuperadas):

- Falta no canal de comunicação
- Falta no nó (servidor, apenas pode falhar 1 em simultâneo)

Uma falta num canal de comunicação ao enviar uma mensagem de escrita para os servidores levará à inconsistência da informação desse utilizador, dado que um dos servidores não será atualizado.
Uma falta no servidor fará com que este perca a informação nele guardada ou, mesmo que a informação estivesse guardada numa base de dados persistente, enquanto o servidor estivesse em baixo, não sofreria as alterações que pudessem ocorrer, ficando possivelmente com dados inconsistentes.
De acordo com o enunciado, tendo em conta que vamos apenas considerar estas faltas, temos assim uma taxa de cobertura de 100%, um valor que só faz sentido neste contexto académico.

Solução de tolerância a faltas e detalhes do protocolo

Na página seguinte pode ser observada a figura da solução de tolerância a faltas, com um exemplo de interação do protocolo implementado, *Quorum Consensus* (QC).
Nesse exemplo, consideram-se 4 momentos de quando o Cliente carrega a sua conta com 2000 pontos, e que desde a transação anterior a essa, uma das réplicas (Réplica 3) já estava com dados inconsistentes (desatualizados) por uma falta anterior.
Nota inicial: Todas as entidades mencionadas pertencem ao módulo *pts*, Cliente e Réplicas (dos servidores).

- Momento 1: o Cliente faz *read*, pois todos os *writes* implicam que assim seja, de forma a obter o valor atual. Há uma falta de comunicação com a Réplica 2 e a mensagem perde-se.
- Momento 2: o Cliente recebe as respostas ao seu *read* vindas das Réplicas 1 e 3. Estas respostas são inconsistentes e, portanto, o Cliente guarda o valor da que tem o *tag* mais alto, a mais recente (Balance=1000). Isto é feito no *Quorum*.
- Momento 3: o Cliente faz então o *write* com o valor de 3000 (1000+2000) mas a Réplica 3 entretanto falhou e encontra-se em baixo.
- Momento 4: o Cliente recebe as respostas das Réplicas 1 e 2, ACK, que sinalizam apenas a recepção da mensagem com sucesso. A operação é concretizada com sucesso (Quórum) e duas das três réplicas têm o valor correto (a Réplica 3 continua com dados inconsistentes).

A solução tolerou tanto faltas de comunicação como de nós e continuamos com dados corretos numa maioria das Réplicas. O protocolo QC garante a consistência dos dados e a tolerância ao nosso modelo de faltas, pois partindo de um estado consistente alcançamos sempre outro estado consistente, que consiste em ter os dados corretos numa maioria das réplicas. Essa maioria garante que um quórum num *read* retorna o valor correto (atual). Um quórum, peça chave do protocolo QC, obtém-se quando a maioria das Réplicas responde com sucesso a um pedido de um cliente. Na operação de *read*, é então usada a *tag* (um número sequencial) para identificar qual a resposta com o valor mais recente, e que, portanto, será o atual/correto. Na operação de *write*, obter um quórum apenas garante ao Cliente que o valor foi escrito numa maioria dos servidores, indicando que a operação teve sucesso.
De referir que, no contexto deste projeto, é sempre obtido um quórum para qualquer operação, visto que temos um número fixo de réplicas, três, e que, no máximo, existe uma minoria de réplicas em falha em simultâneo (uma), estando sempre duas delas aptas a responder ao Cliente.

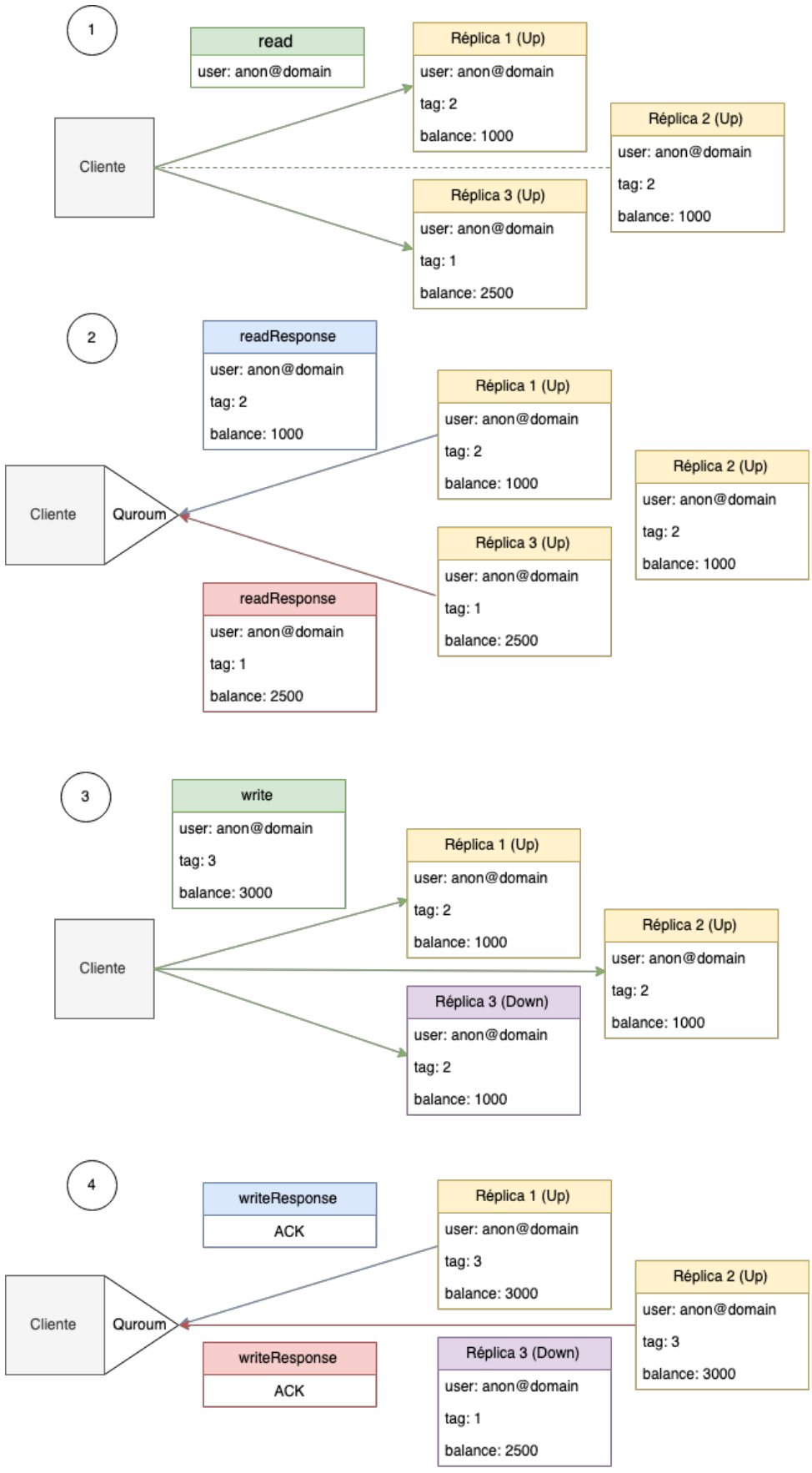

Fig. 1 – Solução de tolerância a faltas

O protocolo QC implica que se definam limites para obter quórum em cada uma das operações (*RT* para *read* e *WT* para *write*). A soma destes dois limites tem de ser superior ao número de réplicas, e, visto que temos sempre três réplicas, definimos apenas uma variável para ambos, *QCneed* com o valor 2. No início de cada operação, a variável de controlo *responseCount* é colocada a zero de forma atómica e é incrementada, também de forma atómica, sempre que o cliente recebe uma resposta positiva de uma réplica. Quando atinge o limite, é obtido um quórum, o cliente deixa de esperar por outras respostas e a operação é concluída com sucesso.
Em termos de implementação, optámos por não criar um *front-end*, pois, tendo apenas um cliente, a solução mais simples é ser ele próprio a gerir o contato com as réplicas. A replicação ativa foi implementada com *polling*, pois o cliente apenas tem de sistematicamente verificar se já obteve respostas necessárias para que obtenha um quórum e, no caso do *read*, analisar o conteúdo dessas respostas.
Criámos duas Classes internas no cliente (*ReaderThread* e *WriterThread*) que estendem a interface *Runnable*, implementando assim um tipo de thread para cada operação, *read* e *write*, cada um com um método *run* único. Os *ReaderThreads* devolvem a *Account* do utilizador, que contém o seu Balance e a *tag*. Os *WriterThreads* apenas sinalizam o sucesso da operação na réplica. Assim, temos então os métodos *readQuorum* e *writerQuorum* no cliente, que começam sempre por atualizar sempre a lista de réplicas disponíveis e gerem a existência dos threads necessários para lhes comunicar a operação.

Optimizações/Simplificações

A simplificação inicial foi no protocolo QC, sendo que o nosso *tag* é apenas o número de sequência pois só possuímos um cliente, não havendo assim necessidade de o identificar.

Foi implementado um sistema de trincos para assegurar escritas atómicas nos registos dos utilizadores. Apesar de não ser testada na demonstração, no caso hipotético de termos vários pedidos concorrentes, assegura-se assim a consistência dos dados e das chamadas às funções de *read* e *write*.

Finalmente, foi também implementada uma cache que, por vezes, nos permite evitar enviar chamadas de *read* às réplicas. No fim de cada operação *read* ou *write*, é criada nessa cache uma entrada registando o *email* do utilizador, o valor do seu saldo atual e a *tag*. A cache tem a possibilidade de guardar cinco registos, que é feito seguindo uma lógica de *LRU* (*Least Recently Used*), ou seja, caso a cache já esteja cheia, reescreve o registo não utilizado há mais tempo. Quando o Cliente recebe uma nova operação, seja *read* ou *write* (que começa com um *read* também), primeiro consulta a cache e, se tiver um registo para o utilizador em questão, não envia *read* para os servidores, simplesmente usa o valor da cache. Caso a operação fosse *read*, está concluída, sem necessidade de comunicar com as réplicas, devolve esse valor. Caso a operação fosse *write*, usa esse valor como se fosse o valor retornado pelo *read* inicial obrigatório, comunicando apenas com as réplicas para enviar o novo valor do saldo. De mencionar que, em ambas as operações, a cache apenas é atualizada no fim, ou seja, o *read* inicial obrigatório antes do *write* não atualiza a cache.