Relatório da 2ª Entrega

Projeto *ForkExec*

Grupo - T08

GitHub:
https://github.com/tecnico-distsys/T08-ForkExec

87641 Carolina Carreira

87691 Miguel Barros

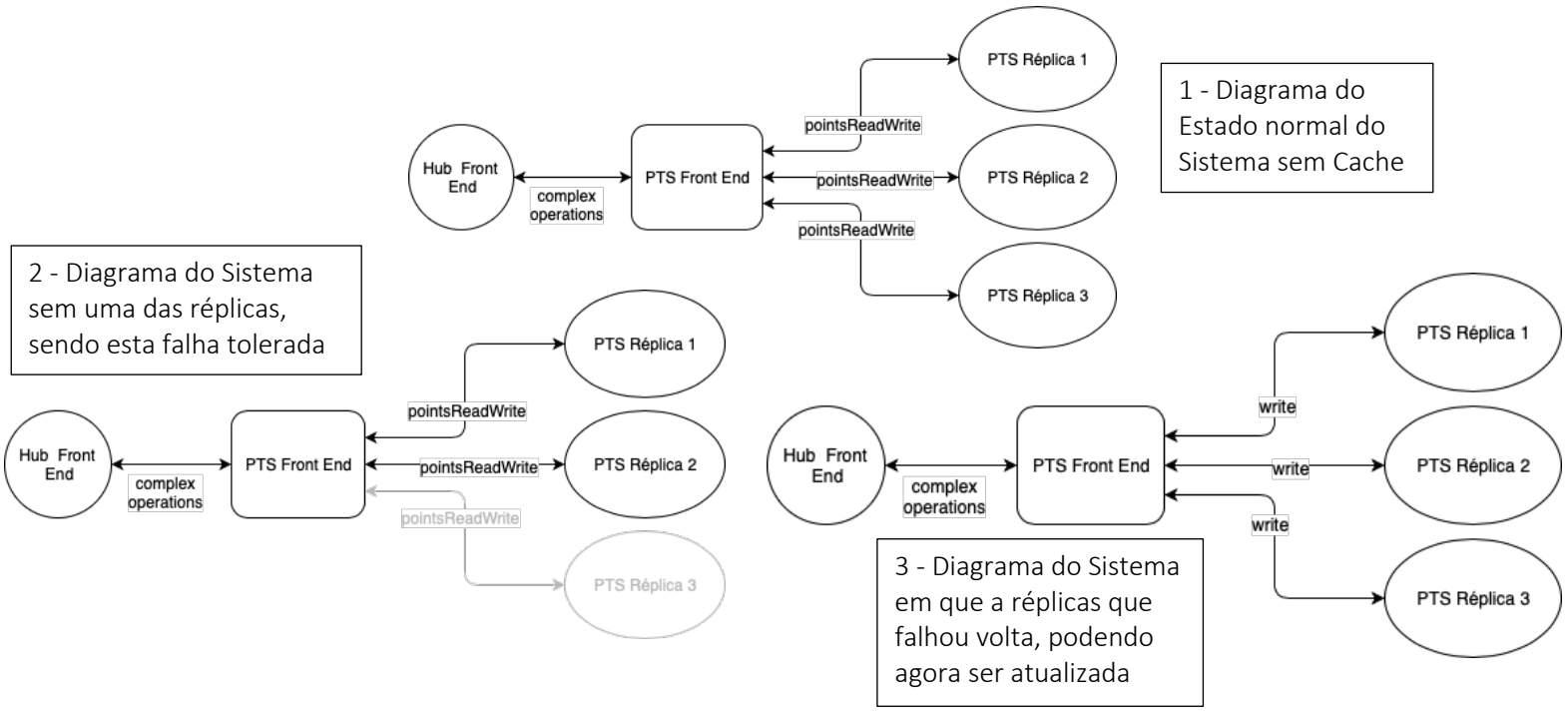
Sistemas Distribuídos
2º Semestre
2019

1. Modelo de Faltas

O sistema é assíncrono e a comunicação pode omitir mensagens. Não há garantia de receção FIFO.
Existem N gestores de réplica e N é constante e igual a 3 (para a demonstração).
Para a nossa implementação são necessárias pelo menos 3 réplicas para tolerar 1 falha de réplicas (devido ao *thresholds* escolhidos, ver seção 4). Como nós temos 3 réplicas, vamos tolerar 1 falha de réplica (na demonstração).
No máximo, assumimos uma minoria de gestores de réplica em falha em simultâneo. Mais do que isto é uma falha catastrófica que não é tolerada.
Os gestores de réplica podem falhar silenciosamente, mas não arbitrariamente.

- Qualquer servidor pode ter uma falta por paragem, enquanto que a rede omitir ou atrasar mensagens.

2. Figura da solução de tolerância a faltas



NOTA: Neste diagrama não estamos a implementar a cache pois na demonstração esta vai estar desligada durante o teste de tolerância a faltas.

3. Breve explicação da solução

O Hub em vez de comunicar diretamente com o PTS comunica com um Front End (FE) e este sim faz a gestão das réplicas do PTS.

Réplicas aceitam apenas operações de leitura ou escrita, por isso PTS FE faz a 'tradução' de operações complexas (exem. incremento) do Hub em operações de leitura e escrita para as réplicas.

Caso uma das réplicas falhe o FE continua a comunicar com as outras duas. (Diagrama 2) Sendo esta falha tolerada pela nossa solução.

4. Descrição de otimizações/simplificações

No protocolo QC original a tag tinha um Cid.
Na nossa solução a tag não necessita de Client ID pois só há um cliente, o Hub.

No protocolo QC todas as escritas são sincronizadas.
Na nossa solução apenas as escritas na mesma 'conta' são sincronizadas. Devido a isto não precisamos de implementar a variante *Write Back* do protocolo QC.

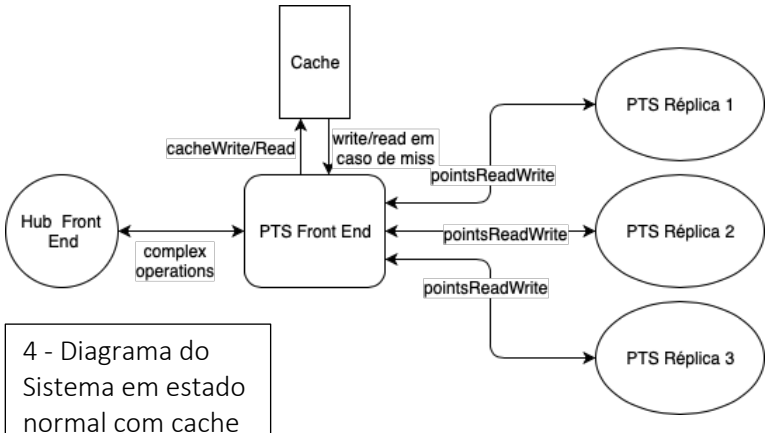
Verificações são efetuadas no Front End na nossa solução, as operações de *read* e *write* assumem que os argumentos estão corretos. Tomamos esta decisão para simplificar nosso código removendo a necessidade de SOAP faults.

Removemos a excepção EmailAlreadyExistsFault pois na nossa implementação, só com *read* e *write,* não temos maneira de saber se o email já está registado. Esta impossibilidade deve-se ao facto de que se o *write* for chamado com um email que já existe este não falha, nem lança excepção, apenas adiciona a conta. O mesmo acontece com o *read*.

Adicionamos uma cache *Write Back*, *Write Allocate* com política de substituição *Round Robin.*
A cache tem a seguinte estrutura:

Valid	*Bit Dirty*	*Balance*	*Mail*
...	...	...	...

Uma operação de leitura no FE procura primeiro na cache, caso não encontre na cache vai então aceder às réplicas atráves do FE e atualizar a cache.
Write Back na nossa implementação é escrever nas réplicas o valor retirado da cache.



4 - Diagrama do Sistema em estado normal com cache

5. Troca de Mensagens

Cada réplica, para cada registo guarda o valor e uma tag.
A tag corresponde ao número de sequência de escrita que deu origem ao registo. Quanto maior a tag mais recente o registo.

A cada leitura o FE:
- envia read() para todos
- espera que RT(*read threshold*) respondam
- escolhe resposta com maior tag

A cada escrita o FE:
- executa uma leitura para obter a maxTag e para obter valor atual da instância
- envia write com newTag = maxTag + 1
- espera WT*(write threshold)* ACKS
- retorna valor escrito ao cliente

Para todas estas operações o:

RT(*read threshold*) = 2

Porque no nosso sistema há mais leituras que escritas.

WT*(write threshold)* = N - RT + 1

Porque RT + WT tem de ser maior que N. Garantindo que uma leitura recebe sempre o valor mais atualizado.

sendo:

- N o número de réplicas;

Dado que temos 3 réplicas o nosso WT = 2

Com este RT o nosso sistema só tolera uma falta (independemente de N>=3), caso quiséssemos tolerar mais poderíamos faze-lo aumentando o RT, sacrificando assim a eficiência da operação de *read.*

Nas operações do FE efetuamos chamadas assíncronas com *callback* a todas as réplicas, em seguidas esperamos RT/WT respostas das réplicas.

O FE PTS suporta as operações (para mais detalhes ver procedimento de leitura e escrita do FE acima descrito):

- **activateUser** - ativa uma conta de email
  faz um Read
- **add/spendPoints** - carregar a conta com pontos
  faz um Read e um Write
- **accountBalance** - obter pontos de uma conta
  faz um Read