Universidade do Minho

escola de engenharia

departamento de informática

Universidade do Minho
Mestrado Integrado em Engenharia Informática - 4º ano

Sistemas Distribuídos Confiáveis

Ano Letivo de 2016/17

Tolerância a Faltas

Controlo de Tráfego Rodoviário

Helder Novais - A64378
João Fernandes - A64341
Paulo Sousa - PG27774

1 de Maio de 2017

Conteúdo

1 Introdução

A gestão de tráfego ferroviário, assim como diversas outras tarefas críticas de gestão, é normalmente apoiada por ferramentas específicas de *software*. No âmbito da disciplina de *Sistemas Distribuídos Confiáveis* foi proposto o desenho e conceção de um sistema que implemente um conjunto de funcionalidades base e que, ao mesmo tempo, garanta que o serviço é tolerante a eventuais faltas nos sistemas remotos.

Neste contexto foi desenvolvido o *RailManager*[1]: um par de aplicações cliente/servidor que disponibiliza o sistema tolerante a falhas de gestão a falhas:

access(r, s, c) determina se a composição **c** pode aceder ao segmento **s** da linha **r**;

enter(r, s, c) assume que a composição **c** entrou no segmento **s** da linha **r**;[2]

leave(r, s, c) assume que a composição **c** saiu do segmento **s** da linha **r**;

getRails devolve um mapeamento entre o nome da linha ferroviária e o seu número de segmentos;

getPositions(r) devolve um mapeamento entre o número do segmento e as composições lá presentes de uma linha **r**;

getAlarms devolve os alarmes gerados no sistema devido à entrada indevida num segmento.

getRails devolve todas as linhas ferroviárias disponíveis no sistema.

O sistema foi desenvolvido na linguagem de programação *Java* e utiliza o *toolkit* de comunicação em grupo *Spread*. Na utilização deste último foi aproveitada a sua funcionalidade de serviço de *Group Membership* para implementar um protocolo de replicação ativa.

2 Protocolo de Replicação

Para que o serviço *RailManager* garanta uma alta taxa de confiabilidade, é necessário que este seja oferecido através de diferentes servidores, e que estes ajam em conjunto segundo um protocolo de replicação.

O protocolo de replicação implementado foi o de replicação **ativa**: os servidores participantes recebem os pedidos dos clientes pela mesma ordem, processam-nos de forma independente e, finalmente, respondem ao cliente. Esta abordagem à replicação mostrou-se mais vantajosa, pois todas as operações são determinísticas, podendo então ser aproveitadas características como a transparência, aos olhos do utilizador, das falhas das réplicas, e

[1] Disponível em `https://github.com/prsousa/rail-manager-fault-tolerant`.

[2] O valor de retorno é *false* caso seja gerado um alarme.

a maior rapidez na obtenção de uma resposta pelo serviço, que é fundamental em sistemas críticos de tempo real.

Existe também a vantagem do controlo não ser centralizado. Na replicação passiva os servidores de *backup* podem demorar bastante tempo até que detetem uma falha *crash-stop* do servidor primário, sendo, consequentemente, a recuperação do serviço mais lenta do que com replicação ativa, onde as restantes réplicas continuam a funcionar normalmente.

3 Servidor

Partindo dos exemplos desenvolvidos durante as aulas práticas foi desenvolvido um servidor *single-threaded* que implementa um protocolo de replicação ativa e que responde a pedidos realizados por aplicações cliente que simulam composições ferroviárias.

A comunicação entre as aplicações cliente e servidor e a própria comunicação entre os vários servidores ativos é feita através do *toolkit Spread*.

Cada servidor, durante a sua atividade, tem a si associado o estado que conterá a informação acerca das linhas, os respetivos segmentos, e informação acerca das composições que lá se encontram num dado momento.

Após a sua inicialização, o servidor verifica se o seu estado permite o processamento de pedidos recebidos. Caso este não tenha sido carregado pelo utilizador, irá esperar até receber o estado atual do sistema vindo de um outro servidor. Enquanto o servidor não se encontra num estado coerente, todos os pedidos recebidos são armazenados pela ordem pela qual foram enviados, para que sejam processados posteriormente.

A entrada de um novo servidor despoleta uma mensagem de *Membership Info* que é entregue a todos os outros servidores, informando-os do acontecimento. Assim, estes enviam o seu estado atual ao novo servidor para que possa também participar na resposta aos pedidos.

4 Cliente

A aplicação cliente desenvolvida, para além de disponibilizar uma interface básica de acesso ao serviço, permite a execução de testes automáticos de performance e de validação. Nestes testes é possível definir tanto a duração dos testes, como o número de *threads* cliente.

Antes de executar o teste automático, e para validar a integridade do sistema, são verificados quantos alarmes existem no momento, sendo que no decorrer do teste são contabilizados quantas chamadas à função *enter* resultaram na criação de um alarme. No final do teste o número de alarmes que o sistema remoto agora devolver deve coincidir com o número de alarmes inicial somado com o número de alarmes gerados.

Sendo que o programa automático de testes apenas aciona o método *enter* se o método *access* correspondente tiver sido aceite, apenas são gerados alarmes quando existem várias *threads* cliente a executar concorrentemente. Isso deve-se à possibilidade de dois pedidos

de *access* a um mesmo segmento (ou a um seu vizinho) serem atendidos, e só num terceiro ser feito um *access*, fazendo com que o último provoque um alarme.

5 Testes e Resultados

5.1 Desempenho do par cliente/servidor na mesma máquina

Para avaliar o desempenho da solução sem a influência da latência acrescida do acesso à rede foram realizados vários testes de performance dentro da mesma máquina.

5.1.1 Diferentes números de clientes e de réplicas

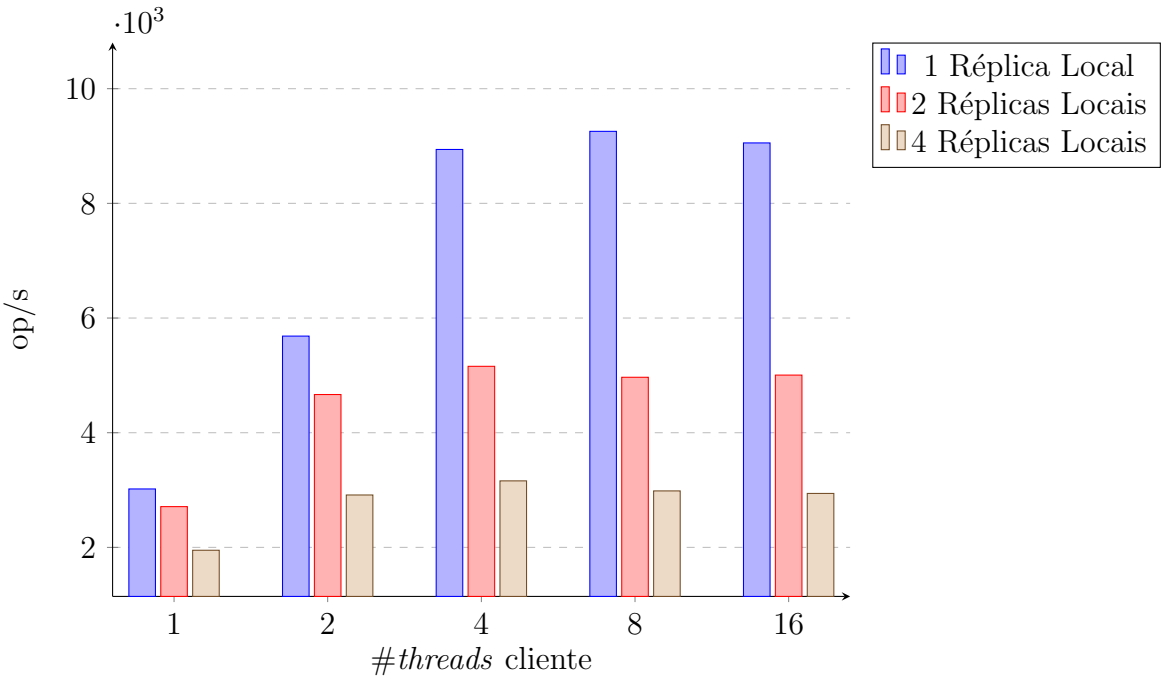

Figura 1: Teste de performance variando o número de réplicas ativas na máquina local

Neste teste verifica-se que o aumento do número de réplicas leva a uma degradação de performance do serviço. Isto deve-se ao aumento do número de mensagens a circular, mas, sobretudo, deve-se ao aumento do esforço que é necessário para preservar as propriedades garantidas pelo *toolkit* de comunicação em grupo perante o aumento do número de réplicas.

Verifica-se também que, com o mesmo número de réplicas, a performance aumenta quando se aumentam o número de clientes até 8[3].

[3]coincidindo com o número de *threads* físicas suportadas pelo processador em teste.

5.1.2 Simulação de falhas e recuperações das réplicas

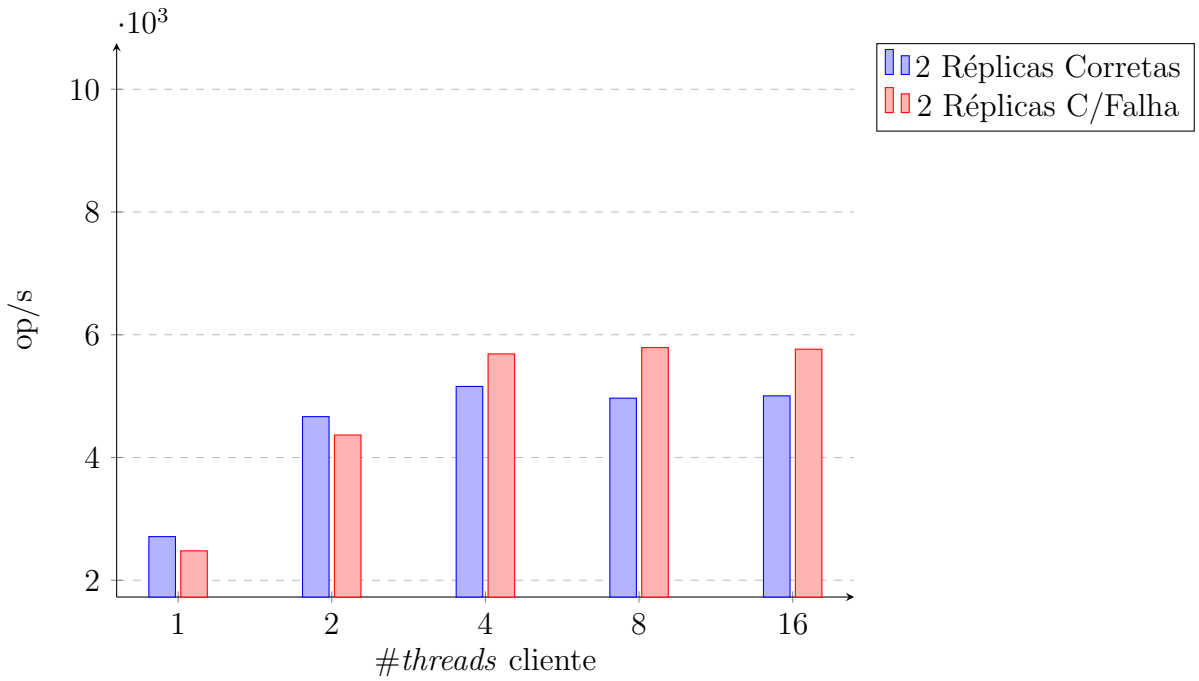

Figura 2: Teste de performance simulando a falha de uma das réplicas

Tal como esperado, mesmo com a falha e posterior recuperação de uma réplica na máquina local, o desempenho não se alterou significativamente face ao comportamento normal. Para números de *threads* cliente superiores a 2, o desempenho até aumentou, sugerindo que a falha da segunda réplica até beneficiou o sistema (talvez pelo protocolo de replicação ter ficado mais ágil ao ter apenas uma réplica).

5.2 Desempenho do par cliente/servidor em várias máquinas

Para avaliar o impacto da latência inerente ao acesso à rede e até do diferente poder de computação de um dos nós, o serviço foi executado e testado utilizando máquinas ligadas em rede.

Uma delas é a mesma que executa a aplicação cliente, ligada à rede através de conetividade *wi-fi*, e a outra é um *Raspberry Pi 2 Model B*, ligado à rede através de *ethernet*.

5.2.1 Solução local vs. Solução em rede

Figura 3: Teste de performance utilizando uma máquina ligada em rede (*RaspberryPi 2 B*)

Quando o sistema opera utilizando máquinas ligadas em rede, a performance é severamente penalizada. A latência de comunicação entre máquinas é evidente e quando uma delas tem recursos de computação limitados, o preço a pagar ainda é mais elevado.

É percetível que o *bottleneck* é a comunicação em rede, pois o tempo médio da execução de um pedido manteve-se quase constante à medida em que o número de *threads* clientes ia aumentando, contrariamente ao que se passava nos testes apresentados na Figura 1 (que originava o degradamento de performance).

5.2.2 Simulação da falha e recuperação da réplica local

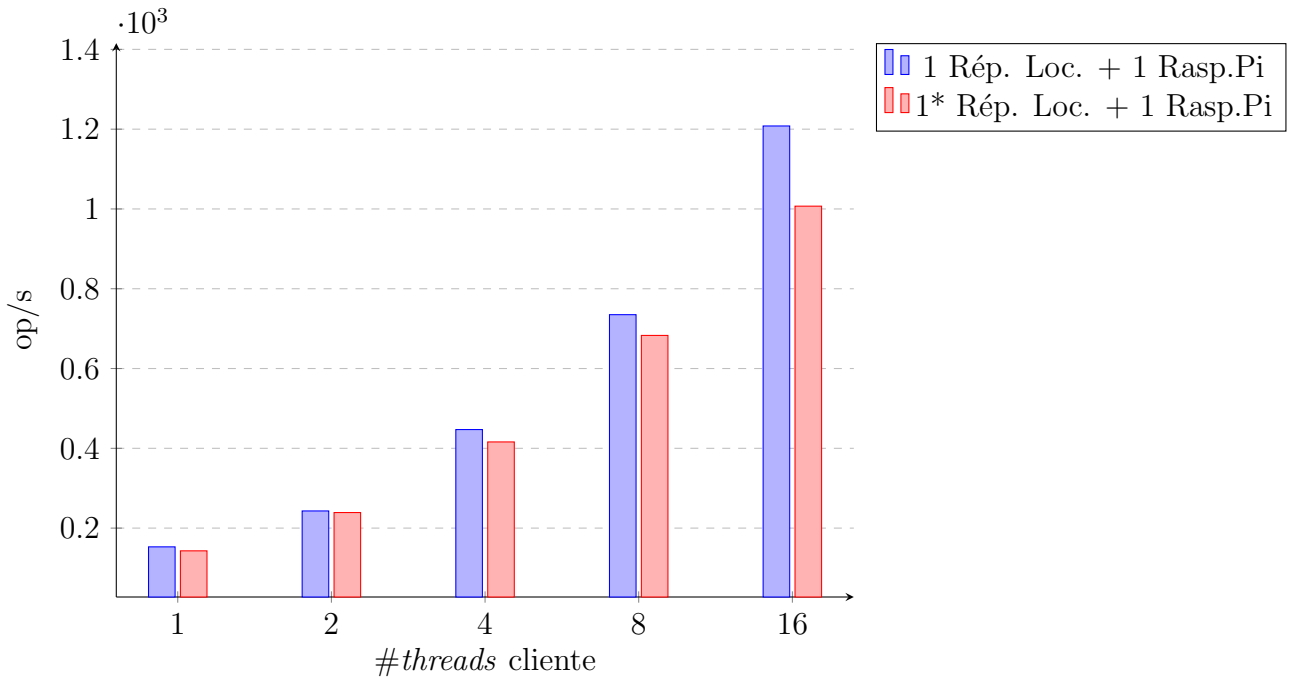

Figura 4: Teste de performance introduzindo a falha da réplica local

Quando a réplica local falha e, consequentemente, deixa de responder aos pedidos, a performance global do sistema degrada-se. Isto era esperado pois agora todos os pedidos são respondidos pela máquina remota que, ainda para mais, tem recursos mais limitados. Esta degradação aumenta com o aumento do número de clientes.

6 Problemas

Um dos problemas encontrados no desenvolvimento do sistema foi relacionado com o *toolkit Spread*. Caso as mensagens sejam longas e ultrapassem o limite, o *Spread* lança uma exceção, impossibilitando o envio da informação entre cliente/servidor e servidor/servidor. Este problema verifica-se quando o sistema tem já um número muito elevado de alarmes gerados, cuja representação é uma lista demasiado extensa de *strings*.

Para o resolver pode desenvolver-se mais uma camada de *middleware* que faça a divisão/junção de partes de mensagens, ou alterar-se a representação dos alarmes, deixando de parte informação obsoleta.