Fundamentos de Sistemas Distribuídos

Trabalho Prático

-

Mestrado em Engenharia Informática
Universidade do Minho
Relatório

Grupo	
PG41080	João Ribeiro Imperadeiro
PG41081	José Alberto Martins Boticas
PG41091	Nelson José Dias Teixeira

3 de Janeiro de 2020

Resumo

Este trabalho prático tem como objetivo a implementação de um sistema de troca de mensagens com persistência e ordenação. Para tal, à semelhança do que foi feito durante as aulas, utilizámos a linguagem *Java* (que é orientada aos objetos) por forma a tomar partido de algumas classes já existentes para este tipo de problemas, como por exemplo a *framework Atomix*. De forma geral, o sistema pretendido pode ser descrito como semelhante ao *Twitter* com alguns requisitos extra de correção.

Conteúdo

Capítulo 1

Introdução

Neste projeto é requerida a elaboração de um sistema distribuído, semelhante ao *Twitter*, baseado na troca de mensagens. Este sistema deve satisfazer alguns requisitos que foram impostos no enunciado deste trabalho, que se listam de seguida:

- o sistema deve incluir um conjunto de servidores que se conhecem todos entre si. Estes não devem ter qualquer interação direta com o utilizador. Admite-se ainda a possibilidade de cada um dos servidores ser reiniciado, devendo garantir que o sistema continua operacional depois de todos os servidores estarem novamente a funcionar;
- o sistema deve incluir clientes que se ligam a qualquer um dos servidores. Admite-se também que o cliente possa ser reiniciado e posteriormente ligado a um novo servidor;
- admite-se que tanto os clientes como os servidores possam fazer uso de memória persistente;
- o cliente deve incluir uma interface rudimentar para interagir com o sistema. Nesta deve-se incluir as seguintes funcionalidades:
  - permitir a publicação de uma mensagem etiquetada com um ou mais tópicos;
  - indicar qual a lista de tópicos subscritos;
  - obter as últimas 10 mensagens enviadas para os tópicos subscritos.
- o conjunto de mensagens obtido por cada cliente em cada operação deve refletir uma visão causalmente coerente das operações realizadas em todo o sistema, por esse ou outros utilizadores.

Capítulo 2

Implementação

O sistema implementado tem como requisito a inclusão de um conjunto de servidores que se conhecem todos entre si e que não devem ter qualquer interação direta com o utilizador. Por isso, foi criada uma interface (***Client.java***) que trata de suavizar essa interação e cumpre os restantes requisitos apresentados anteriormente.

Foi importante definir algumas diretrizes antes de se passar à implementação do sistema. Vamos agora enumerar algumas das decisões tomadas, que serão posteriormente explicadas:

- Cada servidor contém toda a informação do sistema;
- Existe *total order* entre os servidores;
- Os pedidos dos clientes são de dois tipos (bloqueantes e não bloqueantes):
  - Os **pedidos bloqueantes** são: enviar mensagens (*tweets*) e subscrever tópicos.
  - Os **pedidos não bloqueantes** são: ver mensagens (*tweets*) e ver a lista dos tópicos subscritos.
- O cliente não usa memória persistente, ao contrário dos servidores, onde são guardados todos os dados;
- Os servidores utilizam o protocolo *two-phase commit* para garantir a consistência e a coerência do sistema.

2.1 Servidor

2.1.1 Inicialização

Quando é iniciado, o servidor procura uma base de dados. Se encontrar uma e o respetivo ficheiro de *logs*, assume essa mesma base de dados e tenta aplicar as transações pendentes no ficheiro de *logs*, caso exista alguma. Caso não encontre uma base de dados, assume que está a ser criado um novo sistema (*cluster*).

Para a inicialização do primeiro servidor, deve ser executado o ficheiro ***Twitter.java***, com a indicação do endereço *IP* e da porta. Os restantes servidores devem ter a mesma indicação, para além da indicação do endereço *IP* e da porta de um outro servidor já inicializado, por forma a poderem juntar-se a este.

Aquando da criação de novos servidores, que não o primeiro, é realizado um *two-phase commit* para garantir que todos os outros servidores já criados estão a par do novo membro do *cluster*. Isto permite que sejam adicionados novos servidores mais tarde.

2.1.2 Pedidos bloqueantes vs. Pedidos não bloqueantes

Os pedidos que o cliente pode fazer a um servidor do sistema foram divididos em dois tipos:

- bloqueantes - enviar mensagens e subscrever tópicos;
- não bloqueantes - ver mensagens e ver a lista de tópicos subscritos.

Um pedido bloqueante implica que o servidor tenha de comunicar o mesmo aos restantes servidores, iniciar um *two-phase commit* e só depois responder ao cliente: afirmativamente quando todos os servidores estão disponíveis para receber a atualização ou negativamente, caso contrário.

Um pedido não bloqueante não implica que o servidor tenha de comunicar com os restantes servidores, podendo desde logo responder ao cliente.

O que acontece se um dos servidores tiver uma falha e não tiver ainda sido reiniciado, quando outro recebe um pedido? Se o pedido que o servidor recebe for bloqueante, então o mesmo fica bloqueado à espera que o servidor que falhou seja reposto. Se o pedido for não bloqueante, então o servidor responde de imediato, uma vez que não põe em causa a coerência do sistema, visto que não há possibilidade de estar a haver progresso.

2.1.3 Coerência e Ordenação

Os servidores foram implementados com a ideia de *total order* em mente. Para isso, cada servidor possui um contador, que é incrementado quando uma mensagem é enviada e, quando uma mensagem é recebida, é atualizado para o valor seguinte ao máximo entre o seu contador e o do outro servidor. Isto garante que a resposta a pedidos não bloqueantes efetuados ao mesmo tempo em servidores diferentes é igual, quer em termos de conteúdo como em termos de ordenação, visto que os servidores aplicam sempre a transação com o valor do contador mais baixo, utilizando os endereços para desempates.

O facto de a ligação entre um servidor e um cliente utilizar o protocolo *TCP*, permite uma visão causalmente coerente (*causal order*), uma vez que é impossível que uma mensagem "ultrapasse"outra. Ou seja, se um cliente envia duas mensagens para o servidor, a primeira chegará ao destino em primeiro lugar e, portanto, será processada em primeiro lugar, garantido a manutenção da ordenação dos pedidos do cliente.

2.1.4 Gestão dos dados

A informação sobre o sistema, como os tópicos subscritos por um utilizador ou as mensagens partilhadas pelo mesmo, é guardada em todos os servidores. Cada servidor tem uma base de dados, um objeto *DBHandler* (definida no ficheiro ***DBHandler.java***), que contém a seguinte informação:

- um conjunto (*HashSet<Address>*) com os endereços de todos os outros servidores;
- uma lista (*ArrayList<Tweet>*) de todos os *tweets* (mensagens enviadas por utilizadores), sendo que um *Tweet* é composto por uma *string username* (que identifica um utilizador), outra *string content* (com o conteúdo da mensagem/*tweet*) e ainda um conjunto (*HashSet<String>*) de tópicos;

- um mapa (*HashMap<String, ArrayList<Integer> >*) de tópicos para uma lista de inteiros, sendo que esta lista de inteiros corresponde aos índices dos *tweets* com esse tópico;
- um mapa (*HashMap<String, HashSet<String> >*) de *usernames* para um conjunto de tópicos subscritos (pelo utilizador identificado por esse *username*).

Deste modo, os servidores têm sempre, em memória volátil, toda informação existente, não havendo assim o atraso da leitura de disco, apesar de que, depois de cada atualização, o novo estado da base de dados é guardado em disco, de modo que seja possível recuperar esta informação em caso de falha. De realçar que os objetos *DBHandler* de cada servidor devem ser iguais para todos os servidores, uma vez que representam toda a informação do sistema.

2.1.5 *Two-phase Commit*

Sempre que uma operação bloqueante é despoletada, todos os servidores recebem essa informação e, caso tudo esteja em conformidade, prosseguem à gravação da informação. Para isso, é usado o protocolo *two-phase commit*. Portanto, todos os servidores inicializam um objeto *TPCHandler*, definido no ficheiro ***TPCHandler.java***, que guarda a seguinte informação:

- os endereços dos outros servidores, para comunicar com os mesmos;
- a base de dados (objeto *DBHandler*), para ser alterada;
- um contador, para garantir *total order*;
- dois *logs* (um de servidor e um de coordenador), que permitem reiniciar uma transação no caso de um servidor falhar a meio da mesma;
- um mapa com todas as transações em que o servidor é coordenador, tanto as já concluídas (para o caso de um servidor perguntar o estado de uma delas) como as que estão em curso (guardando os servidores que já as aceitaram, para mais tarde efetivá-las);
- um mapa de endereços (de cada um dos outros servidores) para outro mapa de inteiros para pares estado da transação/transação (*HashMap<Address, TreeMap<Integer, Pair<TPCStatus, TwoPhaseCommit> > >*), que corresponde às transações por completar em cada servidor (não-coordenador).

Mas como é controlado o *two-phase commit*? Neste sistema, por não ser implementada tolerância a faltas (uma vez que não é objeto de estudo desta unidade curricular), foi decidido, por razões de simplicidade, que, numa transação, o coordenador é sempre o servidor que recebe o pedido e, consequentemente, inicia a mesma. Assim, este é quem controla o *two-phase commit*.

O coordenador envia a informação a todos os servidores (incluindo ele próprio) e pergunta se todos podem guardar a informação. No caso de todos os servidores darem resposta positiva, o coordenador indica-lhes que devem guardar a informação na base de dados (memória persistente), sendo a transação bem sucedida.

Em cada fase do *two-phase commit*, é guardado num ficheiro de *log* o passo que está prestes a ser efetuado nesse servidor. Para isso, utiliza-se a classe *SegmentedJournal* do *Atomix*, que permite manter um *log* ordenado em disco. Desta forma, no caso de um servidor ser desligado a meio de uma transação, quando este for reiniciado poderá voltar ao estado em que se encontrava, ao aplicar as alterações presentes no ficheiro de *logs*. A única desvantagem desta abordagem é que, há medida que o tempo de

execução do servidor aumenta, o ficheiro de *logs* vai crescendo, levando a um tempo de inicialização maior. Uma solução aplicável seria limpar o mesmo periodicamente.

Tal como os pedidos bloqueantes, a junção de um servidor também inicia um *two-phase commit*, por ser uma operação de grande relevância (a partir daqui todas as operações bloqueantes devem passar por este novo servidor) e que implica a alteração da base de dados dos servidores. Se o *two-phase commit* for bem sucedido, o novo servidor recebe todas as informações do sistema, por forma a inserir-se no mesmo e poder dar uma resposta consistente aos clientes.

Importa ainda referir que foi necessário definir um tipo diferente de *two-phase commit*, os *heartbeats*. Estes surgiram da necessidade de, como abordado nas aulas, forçar os servidores a comunicar, dado que, se um deles parar a comunicação, o sistema não poderá prosseguir como um todo. Assim, criámos os *heartbeats*, que contêm o contador de *total order* do servidor que o envia, e podem ser requisitados por qualquer um dos servidores a todos os outros. Ou seja, sempre que um servidor conclui que necessita de uma mensagem de outro para poder prosseguir, envia um pedido de *heartbeat* a esse servidor, que lhe responderá com a mensagem descrita acima.

2.2 Cliente

Um cliente pode ser ligado a qualquer servidor, sendo que pode ainda desconectar-se e ligar-se a um outro servidor, sem perder os seus dados ou a ordenação dos mesmos. Para ser identificado, o utilizador fornece o seu *username* quando se conecta ao servidor. Se o *username* não existir na base de dados, é criado. Se já existir, assume as informações já existentes.

Não foi implementada autenticação de utilizadores, por não ser relevante para a prática e demonstração das capacidades adquiridas nesta disciplina, pelo que não existe qualquer *login* e/ou registo de utilizadores.

Com o *two-phase commit* e o *total order*, é assegurado que as respostas que o cliente recebe refletem uma visão consistente do estado de todos os servidores.

2.2.1 Interface

Um utilizador pode tomar 6 ações através da interface que lhe é disponibilizada:

- *Tweet* - publicar uma mensagem (*tweet*);
- *Subscribe* - subscrever tópicos, mantendo os que já subscreve;
- *View subscriptions* - visualizar quais os tópicos que subscreve;
- *Last 10 from all subscribed topics* - visualizar os últimos 10 *tweets* sobre os tópicos que subscreve;
- *Last 10 from specific topics* - visualizar os últimos 10 *tweets* sobre um conjunto de tópicos específico, a definir;
- *Exit* - terminar a conexão ao servidor/sistema.

2.3 Funcionalidades extra

2.3.1 Novos servidores

O sistema foi implementado de forma a que possam ser adicionados servidores ao mesmo tempo depois deste já ter sido utilizado, permitindo assim a alteração do número de servidores a qualquer momento no tempo e não ficando o sistema restringido

ao número de servidores com que foi inicializado. Não foi implementada a possibilidade de um servidor se desconectar do sistema, embora a mesma seja fácil (através de um simples *two-phase commit*).

2.3.2 Modularização de código

Quanto à modularização do código, os elementos que compõem este grupo dividiram o mesmo em 3 partes:

- *client*: classe relativa ao cliente e à sua interface;
- *common*: conjunto de classes partilhadas pelo cliente e pelos servidores;
- *server*: classes relativas à implementação dos servidores, incluindo a que trata o ficheiro de *logs*.

Algumas destas classes são específicas a este trabalho, mas algumas delas são suficientemente genéricas para serem aplicadas a qualquer sistema que utilize o protocolo *two-phase commit* e a ideia de *total order*. Há a referir, em particular, as classes *DBHandler* e *TPCHandler*.

Capítulo 3

Testes

3.1 Testes de funcionamento

O sistema foi testado localmente (uma única máquina e vários processos), por forma a garantir que o mesmo funcionava corretamente e que todos os requisitos eram cumpridos.

Para além destes primeiros testes, foram realizados ainda testes adicionais em máquinas remotas (com recurso à plataforma *DigitalOcean*). Estas máquinas estão espalhadas por diferentes cidades, países e continentes, nomeadamente em Londres (Inglaterra), Nova Iorque (EUA), Toronto (Canadá), Amesterdão (Holanda), Singapura e na Índia. Nestas máquinas, foram criados três servidores e alguns clientes. Pudemos conferir que o sistema funcionava corretamente também nestas condições, com os servidores, espalhados pelos locais enunciados, a conectarem-se sem problemas. Confirmamos ainda que os clientes conseguiam executar todas as ações disponibilizadas pela interface de utilizador e que o sistema mantinha a sua consistência e coerência, bem como tinha um comportamento correto em todas as circunstâncias. Para isso, foi simulada a falha de um e de dois dos servidores e o sistema cumpriu o que foi por nós proposto.

3.2 Teste de performance

Realizaram-se alguns testes de performance ao sistema, com números distintos de servidores ativos, localmente e remotamente (servidores espalhados pelo planeta, tal como nos testes de funcionamento). Todos os testes foram feitos usando um cliente, que enviava 50 pedidos por segundo ao servidor ao qual estava ligado e contava quantas respostas recebia. De realçar que todos os pedidos eram de partilhas de mensagens, portanto, bloqueantes.

Tabela 3.1: Número médio de respostas por segundo.

Nº de servidores	**Localmente**	**Remotamente**
1	50	43.4
2	42	37
3	35.3	24

É possível perceber que o aumento do número de servidores implica a diminuição da performance, o que pode ser explicado pelas comunicações que têm de ser estabelecidas entre os servidores, pelo protocolo *two-phase commit*.

É ainda possível constatar que a performance é melhor quando os servidores estão mais perto uns dos outros, nomeadamente na mesma máquina. Isto também é fácil

de compreender, uma vez que se reduz a latência natural que existe com servidores mais distantes.

Obviamente que estes testes de performance dependem sempre das máquinas em que correm os servidores e da ligação que os mesmos têm entre si. É também necessário ter em conta que todos os pedidos são bloqueantes e que os resultados para pedidos não bloqueantes ou mistos seriam sempre diferentes e, provavelmente, melhores.

Capítulo 4

Conclusão

Após a demonstração da abordagem adotada pelo grupo na implementaçao do sistema pedido dá-se por concluída a realização deste projeto. Neste foi possível satisfazer todos os requisitos requeridos bem como acrescentar algumas funcionalidades extra que foram por nós entendidas como relevantes e/ou interessantes.

Foi possível experimentar as dificuldades na implementação de um sistema distribuído, nomeadamente de garantir a coerência, consistência e estabilidade do mesmo. Pudemos também perceber que o protocolo *two-phase commit* permite dar passos importantes no alcance da consistência.

Para além disso, foi possível aprofundar o conhecimento de alguns dos aspetos relativos à componente prática desta unidade curricular, melhorando, assim, a nossa capacidade na implementação de sistemas que abordam esta filosofia computacional.

Apêndice A

Observações

- Biblioteca *Atomix*:
  https://atomix.io/
- Documentação *Java*:
  https://docs.oracle.com/en/java/javase/11/docs/api/index.html