Mestrado Integrado em Engenharia Informática e Computação - Sistemas Distribuídos

Balanceamento de carga em servidores cluster

Autores:
Daniel Filipe Santos Marques - up201503822
João Filipe Lopes Carvalho - up201504875
João Nuno Fonseca Seixas - up201505648
Vicente Fernandes Caldeira Espinha - up201503764

28 de Maio de 2018

Conteúdo

1 Introdução

Nos tempos que correm, muitos problemas do quotidiano ou da ciência moderna podem ser resolvidos através da programação. Contudo, por vezes a complexidade destes problemas requer o processamento remoto em servidores cujo propósito seja somente executar um certo programa de modo a acelerar a sua execução face ao hardware inferior do computador que efetua pedidos ao mesmo. No entanto, mesmo estes servidores possuem limitações quanto ao seu hardware. Daí surgiu a necessidade de repartir a carga de trabalho por várias máquinas/threads de modo a evitar congestionamentos.

O objetivo deste projeto é desenvolver um sistema de balanceamento de carga capaz de distribuir a carga de trabalho uniformemente entre vários computadores/threads a fim de otimizar a utilização de recursos, minimizar o tempo de resposta e evitar sobrecarga. A arquitetura é invisível ao utilizador que, através do programa de cliente, contacta diretamente o load balancer que age como intermediário para a camada de peers que vão efetivamente executar os pedidos.

TODO - Imagem da UI

Este relatório desempenha a função de documentar o sistema desenvolvido e divide-se nas seguintes secções:

- Arquitetura: Esta secção aborda a estrutura da aplicação, evidenciando a interação cliente/servidor, a relação entre o load balancer (servidor) e a camada interna de peers e a comunicação entre nós.
- Implementação: Esta secção aborda os detalhes de implementação importantes para a compreensão do funcionamento do sistema, explicando e justificando as estruturas de dados, algoritmos, solução de concurrência e protocolos utilizados.
- Conceitos Relevantes: Esta secção aborda aspetos importantes para sistemas distribuídos e a sua implementação no projeto, tais como a tolerância a falhas, segurança, escalabilidade e consistência.
- Conclusão: Nesta secção será feita uma retrospeção crítica do trabalho desenvolvido, apontando possíveis aspetos a melhorar.

2 Arquitetura

A estrutura da nossa aplicação está dividida em duas partes. A primeira é o conjunto de peers agrupado pelo *chord algorithm*. A segunda é o Servidor (load balancer) e o Cliente.

O Servidor serve de intermediário entre o Cliente e a rede de peers. Podem haver múltiplos Servidores assim como estes podem estar ligados a qualquer um dos peers. A camada de peers é, essencialmente, invisível ao utilizador, sendo que este comunica apenas com o Servidor que, por sua vez, interage com os peers delegando os pedidos que recebe.

2.1 *Peer (Chord Node)*

Um *Peer* é uma estrutura que corresponde a um nó da rede *p2p chord*. Este pode ser responsável por dois grandes princípios.

Um deles *Worker*, que será um nó presente na rede responsável apenas pela computação de tarefas.

Outro *DatabaseManager*, que será um nó presente numa rede em paralelo com os os *workers*, que será única e exclusivamente responsável, por fazer a gestão da base de dados do sistema. Isto é guardar informação dos utilizadores desde, o nome de utilizador, palavra chave e *tasks* que submeteu.

Estes entre si comunicam e para manterem o bom funcionamento da rede.

Na adição de um novo nó a qualquer uma das duas redes, a Rede de *workers* ou na rede de nó de dados, primeiramente, cada um deles contacta um nó já existente na rede, para saber qual será o seu sucessor. Após saber o seu sucessor, entra em contacto com ele para ele lhe passar todas as responsabilidades pelas quais passará a ser portador. Isto é, no case de ser um nó de gestão de dados, este irá receber os dados guardados no seu sucessor que pertencem ao novo nó. Após esta inicialização estar concluída, o novo nó começa a informar alguns nós presentes na rede que este novo nó existe de modo a facilitar a atualização das *finger tables* do nós. Finalmente o nó dá inicio também aos seus protocolos de estabilidade.

Por protocolos de estabilidade entendemos, um protocolo que mantém constantemente todos os nós com o conhecimento preciso, dos nós, predecessor e sucessor e outro que corre periodicamente uma verificação se a informação que tem guardada na *finger table* está correta.

2.2 Servidor

O Servidor é a estrutura que vai fazer a ligação entre os pedidos do utilizador e a rede *p2p chord*, ou seja, vai servir de intermediário. O Servidor contém dois nós dessa rede (um *WorkerNode* e um *DatabaseNode* de forma a fazer os pedidos necessários. Tem também um *SSLSocket* que é o socket de comunicação com o Cliente.

2.3 Cliente

O Cliente tem uma UI uma vez que é a estrutura que vai interagir com o utilizador. Contém um *SSLSocket* para a comunicação com o intermediário, de forma a fazer chegar à rede *p2p chord*.

2.4 Protocolos de Comunicação

2.4.1 Protocolo *Peer-to-Peer*

Os peers comunicam entre si através do protocolo definido pelo *chord protocol*. Cada peer deve ser capaz de responder a um conjunto de mensagens que permitam o funcionamento do algoritmo, tais como:

- FIND_SUCCESSOR: Enviada encontrar o sucessor de um nó específico dos seus atributos;
- REPLY_FIND_SUCCESSOR: Mensagem de retorno com informação do sucessor de um nó;
- SET_PREDECESSOR: Notifica um sucessor que deve atualizar o seu predecessor para o especificado na mensagem;
- REPLY_SET_PREDECESSOR: Mensagem de retorno do SET_PREDECESSOR, com informação do predecessor anterior;
- GET_SUCCESSOR: Pede o sucessor do nó;
- REPLY_GET_SUCCESSOR: Retorna p sucessor de um nó;
- GET_CLOSEST: Enviada para encontrar o predecessor mais próximo do nó especificado na mensagem;
- REPLY_GET_CLOSEST: Mensagem com informação o nó predecessor mais próximo de um nó;
- GET_PREDECESSOR: Pede o predecessor do nó;
- REPLY_GET_PREDECESSOR: Retorno o predecessor de um nó;
- UPDATE_FINGER: Mensagem de notificação para a atualizar a informação da *finger table* presente perante um certo nó;
- GET_LOGIN_DATA: Pede pela informação de *login* que o nó especificado na mensagem é responsável;
- REPLY_LOGIN_DATA: Retorna informação de *login*;
- GET_TASKS_DATA: Pede pela informação de tarefas que o nó especificado na mensagem é responsável;
- REPLY_TASKS_DATA: Retorna informação de tarefas;
- SAVE_TASK:
- EXECUTE:

2.4.2 Protocolo *Server-to-Peer*

- LOGIN: Mensagem com pedido de *login* a um nó de dados;
- REPLY_LOGIN: Mensagem de resposta ao pedido de *login* perante o servidor ao nó;
- REGISTER: Pedido de registo por parte do servidor ao nó;
- REPLY_REGISTER: Resposta da conclusão do registo na rede de nós ao servidor;
- SUBMIT: Servidor pede para a rede de nós de processamento agendar a execução de uma dada tarefas;

- GET_TASKS: Servidor requer todas as tarefas executadas por parte da rede de um dado utilizador;
- REPLY_GET_TASKS: Resposta ao pedido descrito no ponto anterior;

2.4.3 Protocolo *Client-to-Server*

- LOGIN: Envia o usernamO cliente também comunica com o servidor através de mensagens:e e a password de forma a verificar se o utilizador está registado no sistema.
- REGISTER: Envia o username e password de forma a registar um novo utilizador no sistema.
- TASK: Envia a informação do utilizador(username e password) e da tarefa a executar.
- CONSULT: Envia a informação do utilizador para pedir a informação do estado das tarefas.
- LIST_TASKS: Envia a informação do utilizador para obter todas as tasks do utilizador.
- DELETE: Envia a informação do utilizador e a tarefa a eliminar.

2.4.4 Protocolo *Server-to-Client*

- 200: Significa que a operação correu bem. Pode ter ainda mais informação dependendo do pedido.
- 300: Indica que é uma mensagem de UI, com prints e com escolha de opção num menu.
- 400: Significa que algo correu mal na execução do pedido ou que, por exemplo, já existia o utilizador quando se vai a registar.

3 Implementação

O projeto foi desenvolvido apenas em Java.

A utilização de execução paralela, com recurso a multithreading, requer uma solução que lide com problemas associados a concorrência. Para isso, foram utilizadas estruturas de dados das bibliotecas de Java capazes de, nativamente, eliminar situações causadas por *race conditions*. Destas destaca-se o uso de *ConcurrentHashMap* e *ConcurrentSkipListMap*.

3.1 *ConcurrentHashMap finger_table*

A nossa *finger table*, que está presente em cada nó, contém informação até 32 nós. Isto dás-nos a possibilidade de uma escabilidade de até 2^{32} nós na rede.

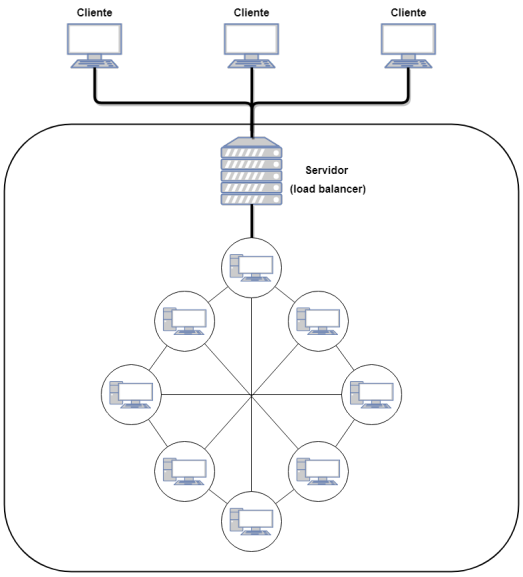

Figura 1: Representação da arquitetura

Como estamos a utilizar números inteiros para a representação dos id's dos nós, temos de controlar problemas como o overflow. Que neste caso até é uma vantagem para nós. Dado que overflow faz com que dê continuidade circular.

3.2 *Client* e *Server*

Na implementação do *Client* e do *Server* foram usadas as bibliotecas *java.security* (para a classe *KeyStore*) e *javax.net.ssl* (para as classes *KeyManagerFactory* e *TrustManagerFactory*) para autentificação de certificados e chaves entre o cliente e o servidor de forma a tornar seguras as comunicações entre estes. Foram também usadas as classes *SSLSocket*, *SSLServerSocket* e *SSLContext* da biblioteca *javax.net.ssl* para criar os sockets de comunicação entre servidor e cliente e vice-versa.

3.3 Troca de Mensagens

A aplicação é conseguida através da contínua troca de mensagens entre cliente-servidor e servidor-servidor. Para isso, e como referido anteriormente, foi criada a Classe Message, sendo esta uma classe Serializable. De forma a tornar o processo troca de mensagens mais simples e eficaz foi implementado um processo de Serialização. Este processo consiste na tradução de estruturas de dados num formato (array de bytes) que possa ser armazenado e reconstruido posteriormente. Para isso, as classes Message, User, Task, ... implementam a

classe Serializable.

```
1   public static boolean send(Socket socket, Message request){
2       if(Constants.DEBUG)
3           System.out.println("SENDING " + request.getType());
4       try {
5           ObjectOutputStream oStream = new ObjectOutputStream(
                socket.getOutputStream());
6           oStream.writeObject(request);
7       } catch (IOException e) {
8           exceptionPrint(e, "[ERROR] Sending message ");
9           return false;
10      }
11      return true;
12  }
```

Cada nó contém um objeto responsável por dar *handle* dos pedidos de mensagens recebidos. Este objeto contém um *ScheduledThreadPoolExecutor* que será responsável por tratar de problemas de concorrências à *pool de threads*.

4 Conceitos Relevantes

Este projeto implementa algumas características importantes no contexto de sistemas distribuídos:

- Tolerância a falhas: Continua analise dos nós vizinhos para assegurar que a rede Chord se mantém estável, assim como a verificação da finger table
- Segurança: São utilizadas chaves e certificados autenticados para segurança na comunicação via SSL entre Cliente e Servidor.
- Escalabilidade: O sistema é escalável dado que, se necessário, novos nós podem ser executados e adicionados à rede para responder a mais pedidos em simultâneo. O algoritmo *p2p chord* consegue atingir escalabilidade sem envolver hierarquia.
- *Stateless*: O Servidor não guarda nenhum estado da comunicação com qualquer Cliente. O Cliente, em cada pedido, envia os dados suficientes necessários à execução do pedido.

5 Conclusão

O sistema cumpre, essencialmente, todos os objetivos e requisitos de funcionamento. Acabou por ser ligeiramente diferente do apresentado na especificação, nomeadamente na parte da rede de Nós, que passou a ser a rede *p2p chord*. Existe aspetos a melhorar que seria o facto de este funcionar pela Internet e não apenas por rede local. Também a comunicação por HTTPS seria ideal se tivesse sido implementada.