Faculdade de Engenharia da Universidade do Porto

Serviço de Backup Distribuído pela Internet

Sistemas Distribuídos

3º ano do Mestrado Integrado em Engenharia Informática

Elementos do grupo:

- ➔ Cláudia Margarida da Rocha Marinho, up201404493, up201404493@fe.up.pt
- ➔ Fabiola Figueira da SIlva, up201502850, up201502850@fe.up.pt
- ➔ Mariana Duarte Guimarães, up201307777, up201307777@fe.up.pt
- ➔ Tiago André Pérola Filipe, up201610655, up201610655@fe.up.pt

27 de maio de 2018

Conteúdos

1. Introdução

O nosso projeto é um melhoramento do primeiro projecto desenvolvido para a unidade curricular de Sistemas Distribuídos, em que implementámos a versão de Internet para o Serviço de Backup Distribuído.

Neste projeto, além de termos implementado os protocolos sobre a Internet, introduzimos segurança, tolerância a falhas e escalabilidade.

2. Arquitetura

2.1 Componentes e interações

Os dois componentes principais do programa são o Peer e o Master Peer, sendo que existem três tipos de interações:

- Peer com Master Peer vice-versa;
- Master Peer com outro Master Peer;
- Peer com Peer.

De forma a explicar os dois primeiros tipos de interações, na tabela seguinte serão descritas as classes utilizadas e quais as suas responsabilidades:

Classe	**Responsabilidade**
PeerServerListener	Trata da receção e envio de todas as mensagens entre o SSLSocket do Peer e o SSLSocket do Master Peer ao qual o Peer está conectado.
ServerPeerListener	Trata da receção e envio de todas as mensagens entre o SSLSocket do Master Peer o SSLSocket do Peer.
ServerToServerListener	Aguarda que outro Master Peer comunique com o ServerSocket pertencente ao Master Peer.
ServerToServerChannel	Trata da receção e envio de todas as mensagens entre os Sockets dos Master Peers.

Na interação Peer com Peer, obrigatoriamente há necessidade de recorrer à comunicação com o Master Peer para obter quais os peers que estão ligados ao sistema. Após isso a comunicação entre Peers é feita através do envio de DatagramPackets utilizando deste modo o protocolo UDP.

O capítulo seguinte descobre mais pormenorizado o conceito de Master Peer e a estrutura definida para eles.

2.2 Master Peers

Considerando que passámos de uma rede local para a Internet, deixou de ser viável usar *Multicast* na comunicação entre os vários *Peers*. Assim sendo, introduzimos

o conceito de *Master Peer* na nossa aplicação para tornar possível esta comunicação entre os *Peers* pela Internet.

No nosso sistema, podemos usar no máximo 3 *Master Peers* que, após criados, ficam à escuta nos ports 3000, 3001 ou 3002 à espera que algum Peer se autentique. A comunicação entre os 3 Master Peers, funciona através do mesmo IP, mas nas portas 3100, 3101 e 3102. Poderíamos ter alterado a implementação e colocar cada Master Peer com um IP diferente, o que obrigaria o Peer a necessitar de conhecer esses 3 IPs, mas decidiu-se que não era relevante e portanto, o Peer apenas necessita de conhecer um IP para se conectar a qualquer um dos três.

Todos os *Peers* criados devem-se registar num dos *Master Peers*, mandando o seu ID e os ports onde se encontra à escuta (correspondentes aos canais MC, MDB e MDR) para o *Master Peer* em que se registou. Assim, quando um Peer quiser comunicar com os outros Peers para executar os protocolos, este terá de pedir ao *Master Peer* a que está ligado informações sobre todos os outros peers ativos.

3. Implementação

3.1 Concorrência

Relativamente à concorrência na execução de tarefas do projeto, sendo que uma das partes principais é um peer estar à escuta de mensagens em três portas diferentes (canal MC, MDB e MDR), optou-se por fazer esses *listeners* em threads diferentes, com o objetivo de quando o peer estiver à escuta de mensagens nunca ficar impedido de executar outras tarefas.
Como existem várias tarefas que necessitam de ser executadas quando alguma mensagem é recebida num canal, houve a necessidade de lançar um *EventHandler*, quando o canal recebe um pacote, inicia uma thread que trata desse pacote recebido. Como se pode verificar na classe *PeerChannel*, pertencente ao package *peer*, na linha de código nº35, que invoca uma thread com a classe *EventHandler* instanciada.

Para além de estar à escuta das mensagens que vários peers enviam através desses canais, o programa também consegue em simultâneo estar a aguardar por pedidos do cliente. Sempre que um pedido de protocolo é recebido, esse pedido é resolvido numa nova thread à parte.

Posto isto, não chega apenas uma boa estrutura de threads para o programa ser eficiente, existe também a necessidade de utilizar ferramentas que garantam a execução com sucesso das tarefas das threads. Para isso, no que toca a acesso de memória run-time partilhada entre threads utilizaram-se as seguintes classes da biblioteca concurrent do Java: ConcurrentHashMap e CopyOnWriteArrayList que garantem um acesso thread-safe aos dados. Sobre a escrita da memória não volátil utilizaram-se os métodos synchronized do Java. A solução ideal seria utilizar a classe AsynchronousFileChannel, mas surgiram algumas dificuldades e para o desenvolvimento do trabalho não atrasar optou-se por abdicar desta solução. Essa implementação encontra-se na classe *MetadataManager* pertencente ao package *filemanager*.

Para finalizar, tal como recomendado, evitou-se a utilização do método sleep pertencente à classe Thread e utilizou-se a classe *ScheduledThreadPoolExecutor* pertencente à biblioteca concurrent para se efetuar os timeouts das threads. Quando era necessário executar uma tarefa após um determinado intervalo de tempo, agendava-se essa tarefa para iniciar quando esse intervalo de tempo termina-se. Como exemplos dessa implementação, existem as seguintes classes: *BackupMetadata*, na linha de código nº 34, que executa uma thread após um determinado tempo; Restore, na linha de código nº74, que verifica se um chunk foi restaurado após um determinado tempo.

4. Problemas relevantes

4.1 Segurança

- Autenticação SSL

Uns dos métodos de segurança aplicados nesta implementação, foi autenticação SSL devido a necessidade de garantir que apenas os peers por nós implementados sejam capazes de interagir com o sistema. Na autenticação os peers e os Master Peer têm uma keystore e partilham uma truststore, caso a autenticação se verifique com sucesso o Master Peer irá armazenar as informação do peer, nomeadamente IP e porta. Desta forma todos os Master Peer tem uma lista de peers com respectivos dados.

- Encriptação dos chunks

O outro método de segurança aplicados nesta implementação, foi a encriptação dos chunks, este processo ocorre no *InitiatorPeer*, logo após iniciar o protocolo de backup, onde chunks são encriptados com uma key conhecida apenas pelo *InitiatorPeer*, de forma a que mais nenhum peer ou interferência externa consiga reconstruir os chunks e obter a informação armazenada no respetivo ficheiro.

O processo de desencriptação ocorre ao executar o protocolo restore, e é semelhante ao backup, onde os peers que armazenam os chunks limitam-se a encaminhar o bytearray com a informação encriptada para o *InitiatorPeer* e este por sua vez desencripta os chunks para reconstruir o ficheiro.

4.2 Tolerância a falhas

Para suportar a tolerância a falhas nos Peers, cada um dos Peers envia os seus metadados de X em X tempo para os Master Peers (no nosso caso, de 30 em 30 segundos). Quando o Peer é iniciado, verifica se tem metadados consigo, caso tenha carrega esses dados, caso não tenha pergunta ao Master Peer ao qual está conectado se tem os seus metadados. Em caso positivo, o Master Peer envia esse ficheiro e o Peer carrega as informações, senão o peer é iniciado sem informação nenhuma.

Relativamente à tolerância a falhas no Master Peer, sempre que um Master Peer recebe metadados de algum Peer, para além de guardar esses metadados, também os envia para os outros Master Peers existentes. Assim, se após um Peer fazer backup dos seus metadados, se ele e o Master Peer a que estiver conectado, terminarem e perderem todas as informações, o Peer quando se voltar a ligar, irá conectar-se a outro Master Peer que terá os seus metadados, mesmo que o Peer nunca se tenha ligado a

ele anteriormente. Isto acontece, pois sempre que um Master Peer vai abaixo, os Peers ligados a ele, tentam reconectar-se aos outros Master Peers.

4.3 Escalabilidade

Para resolver o problema de escalabilidade, pode-se criar mais que um Master Peer (até um máximo de 3) para dividir o trabalho de cada um dos servidores. Quando existe mais que um *Master Peer*, os novos peers vão-se tentar ligar de forma aleatória a um dos *Master Peers* que se encontram ativos. Este processo foi implementado de tal forma que se houver pelo menos um *Master Peer* ativo, o Peer conecta-se garantidamente a um deles.

Se um *Master Peer* for abaixo, os peers que estiverem ligados a ele, tentam ligar-se aos outros *Master Peers*. Se um peer não tiver nenhum *Master Peer* disponível para se ligar, termina.

5. Guião

Ao efetuar um pedido de backup não esquecer que a localização do ficheiro tem que ser a pasta "**bin/Peers/PeerDisk1/MyFiles**" neste caso o IniciatorPeer é o 1, este será sempre apagado ao executar o comando "bash compile.sh".

- Na raiz do projecto compilar ficheiros com ➜ **bash compile.sh**;
- Iniciar o RMI com ➜ **bash rmi.sh**;
- Entrar na pasta bin e iniciar três Master Peer com ➜ **java server.Server <ServerId> <porta>**, o ServerId deve ser único e o domínio das portas é [3000,3002], os Master Peers dever estar na mesma máquina;
- Dentro da pasta bin, iniciar três peers (um terminal para cada) para poder efetuar backup com replicationDegree = 2 ➜ **java peer.RunPeer <PeerId> <ServerIp>**, o peerId deve ser único e o SeverIp é o ip da máquina onde estão os Master Peers;
- **IMPORTANTE,** é fundamental que a localização do ficheiro que irá utilizar para fazer backup seja "bin/Peers/PeerDisk<peerId>/MyFiles" ;
- Dentro da pasta bin, iniciar client com ➜ **java client.Client peer<peerId> <protocolo> <fileName> <replicationDegree>**, o argumento replicationDegree é apenas para o backup;
- O sistema continua em bom funcionamento mesmo após a perda de um servidor, com ➜ Ctrl + C;
- Também é possível desconectar um peer, o Master Peer irá removê-lo da sua lista de peers ativos ➜ Ctrl + C;

- Ubuntu (instruções)
  - bash compile.sh
  - bash rmi.sh
  - cd bin
    - java server.Server 1 3000
    - java server.Server 2 3001
    - java peer.RunPeer 1 <ServerIp>
    - java peer.RunPeer 2 <ServerIp>
    - java peer.RunPeer 3 <ServerIp>
    - java client.Client peer1 BACKUP <fileName> <replicationDegree>
    - Ctrl + C ➜ no terminal do Master Peer 1
    - java client.Client peer1 RESTORE <fileName>
    - java client.Client peer1 DELETE <fileName>
    - java client.Client peer3 RECLAIM 64

- java client.Client peer3 STATE
- java peer.RunPeer 4 <ServerIp>
- java client.Client peer1 BACKUP <fileName> <replicationDegree>

6. Conclusão

Com a elaboração deste projeto adquirimos capacidades fundamentais para o bom funcionamento de sistemas distribuídos, uma vez que foi preciso pensar em uma arquitectura e distribuição de classes de forma a atingir o melhor funcionamento possível, elaboração de estratégias que garantisse a consistência na informação armazenada como também dos recursos disponíveis a cada instante, nomeadamente os peer disponíveis.