Projecto 1 – Notas Sobre Concorrência

1. Introdução

Este documento sugere uma possível via para uma implementação altamente concorrente e escalável do Projeto 1. Usa-se como exemplo a implementação do *Chunk Backup Protocol* (mensagem **PUTCHUNK**).

O que vos propomos é que desenvolvam uma implementação gradual do protocolo e, através de iterações sucessivas, melhorem a simultaneidade/concorrência e a escalabilidade da vossa implementação. O que é apresentado nas secções seguintes segue precisamente este método de desenvolvimento gradual.

Este documento não cobre, obviamente, todos as soluções possíveis.
Se se sentirem confiantes no vosso trabalho de implementação com *threads* podem saltar diretamente para a seção 3.

2. Implementação *Single Threaded*

Suposição: nesta implementação considera-se que é apenas necessário executar uma instância do protocolo de cada vez.

No final da implementação desta versão (simplificada) do protocolo, serão capazes de efectuar o *backup* de ficheiros com um único *chunk*, ou com vários deles, desde que tratem do *backup* de um *chunk* apenas depois de terminar o processo de *backup* do *chunk* anterior.

A ideia de base consiste em usar um único *thread* para cada um dos seguintes:
- *peer* iniciador
- *peer* não iniciadores

Simplificação: dado que esta implementação se trata apenas de uma versão preliminar, não há necessidade de medir com precisão o intervalo entre as retransmissões (da mensagem **PUTCHUNK**). Em vez de se medir o intervalo entre retransmissões, mede-se antes a duração do silêncio no meio (canais), ou seja, desde a transmissão do **PUTCHUNK** mais recente ou desde a recepção do **STORED** mais recente.

Nesta versão simplificada do protocolo pode fazer-se uso de *DatagramSocket.setSoTimeout()* e *Thread.sleep()* para medir a passagem do tempo (e implementar tempos de espera), e usar um único *thread*, tanto para o *peer* iniciador como para os outros *peers*.

3. Um *Thread* por Canal *Multicast*, um Protocolo de cada Vez

Para esta versão da implementação, continuamos a assumir que não há execuções simultâneas de diferentes instâncias de protocolo (*Chunk Backup Protocol*), ou seja, a qualquer momento, há no máximo uma instância de protocolo em execução.

A ideia de base da solução presente consiste em empregar um *thread* por canal *multicast* para tratar da recepção das mensagens enviadas para esse canal. Teremos assim o *Control Receiver thread* (que está á

escuta no canal MC) e o *Data Backup Receiver thread* (que está á escuta no MDB). Para o *Chunk Backup Protocol* não é necessário interagir com o canal MDR.

Acrescidamente, cada um dos *threads* mencionados processa as mensagens recebidas, através do respectivo canal, de forma sequencial. Cada *thread* conclui o processamento de uma mensagem antes de iniciar o processamento da mensagem seguinte. Assim, cada um destes *threads* deve executar um *loop* infinito e em cada iteração deste trata de:

- receber uma mensagem no respectivo canal *multicast*;
- processar a mensagem recebida.

O envio de mensagens **PUTCHUNK** pelo *initiator peer* deve ser realizado por um *thread* diferente dos anteriores, i.e. o *thread multicaster*. Desta forma, o *thread multicaster* pode executar *Thread.sleep()* entre o *multicast* de mensagens **PUTCHUNK** consecutivas, implementando o intervalo de espera especificado no guião. Este thread pode ser criado em resposta a uma solicitação do *testClient*, para executar uma instância do *Chunk Backup Protocol*, e pode terminar quando a instância do protocolo acabar de se processar.

Se pretendem implementar a interface entre o *testClient* e o *peer* usando RMI, o *thread multicaster* pode ser o próprio *thread* criado pelo RMI *run-time* para lidar com a operação de *backup* invocada pelo *testClient*.

Caso contrário, ou seja, se pretendem empregar UDP ou TCP para essa interface, será necessário um *thread* adicional para receber os pedidos do *testClient*. Tal como os *Receiver threads*, este *thread* também pode processar os pedidos do *testClient* em série: ou seja, deve concluir o processamento de um pedido antes de iniciar o processamento do próximo (na verdade, se suportasse a execução simultânea de pedidos do *testClient*, violaríamos a suposição, acima expressa, de que existe no máximo uma instância do protocolo em execução a qualquer momento).

Notem que, no *initiator peer*, o *thread multicaster* e o *Control Receiver thread* podem precisar de partilhar informações (i.e. o número de mensagens **STORED** recebidas), para que o primeiro saiba como proceder. Compete-vos assim garantir que na comunicação entre os dois (por exemplo, no acesso aos objectos partilhados para a implementação dessa comunicação) não haja *race conditions* (problemas de concorrência).

Nota: nesta versão da implementação, os *peers* não iniciadores não precisam de processar as mensagens **STORED**. De qualquer das formas, a implementação de uma tal funcionalidade nesses *peers*, não seria assim tão complexa até porque já é algo que deverá ser feito ao nível do *peer* iniciador.

4. Um Thread por Canal M., Múltiplas Instâncias do Protocolo Simultâneas

Nesta versão da implementação considera-se que já deverá ocorrer mais do que uma execução simultânea do protocolo.

Utilizamos a mesma "arquitectura" de *threads* da versão anterior: existe assim um *Receiver Thread* por cada canal *multicast*, e cada um deles conclui o processamento de uma mensagem, antes de iniciar o processamento da próxima.

Embora possa haver mais do que um *thread multicaster* em execução, tanto no colectivo de *peers* como em cada peer individual (ou seja, pode haver vários peers a iniciar *backups* no sistema), cada um desses *threads* participa apenas na execução de uma instância do *Chunk Backup Protocol*.

De forma oposta, os *Receiver threads* podem precisar de processar mensagens relativas a uma instância do protocolo entre o processamento de mensagens relativas a outra instância do protocolo (para poderem lidar com vários pedidos simultâneos do colectivo de *peers*). Os *Receiver threads* devem, assim, empregar um objecto (estrutura de dados para armazenamento e partilha de informação), por instância de protocolo, para manter as informações relevantes para o processamento de mensagens pertencentes a essa instância de protocolo.
Esse objecto deverá ser construído aquando da transmissão/recepção da primeira mensagem da sua instância de protocolo, dependendo se o *peer* é o iniciador (transmissão) ou não (recepção) dessa instância do protocolo. Além disso, o objecto em questão deve ser mantido numa estrutura de dados, partilhada, de onde será recuperado sempre que isso seja necessário para processar cada uma das mensagens subsequentes da sua instância do protocolo de *Backup*. Desta forma, a primeira acção de um *Receiver thread* após receber uma mensagem (do seu canal *multicast*) deverá ser a recuperação (*retrieval*) do objecto (de armazenamento de dados) da instância do protocolo (*Backup*) à qual a mensagem pertence. Após isso poderá então usar as informações armazenadas nesse objeto para processar a mensagem.

Dica: empreguem uma instância da classe *java.util.concurrent.ConcurrentHashMap* para manter os objetos (armazenadores de informação a ser trocada entre *threads*) das diferentes instâncias de cada protocolo (i.e. um *ConcurrentHashMap* por protocolo).

5. Processando Diferentes Mensagens Recebidas Simultaneamente no mesmo Canal

Na versão anterior, as mensagens recebidas através de um determinado canal *multicast* eram tratadas em série.

Para suportar o processamento simultâneo de diferentes mensagens recebidas no mesmo canal, é necessário empregar mais do que um *thread* por canal. A solução típica passa pelo emprego de *Worker threads*: ou seja, empregar um *thread* diferente do *Receiver thread*, para processar as mensagens recebidas. Especificamente, o *Receiver thread* deve lançar um novo *thread* (*Worker* thread) para o processamento de cada mensagem recebida.

5.1. Uma Solução mais Escalável

O problema com a solução anterior é que a criação e o término de *threads* implica algum *overhead*, no qual se incorre uma vez por mensagem. Uma solução mais escalável é usar uma *pool* de *threads*, o que permite evitar a criação de um novo *thread* para processar cada mensagem.

Assim, esta iteração da implementação deverá empregar *thread pools*. Para isso, convém usar a classe *java.util.concurrent.ThreadPoolExecutor*.

6. Não recorrer a *Thread.sleep()*

O recurso a *Thread.sleep()* para implementação de *timeouts* pode levar à existência simultânea de um grande número de *threads* (em espera), cada um dos quais requer alguns recursos, limitando, portanto, a escalabilidade da solução.

Para evitar esta situação, poderão empregar a classe *java.util.concurrent.ScheduledThreadPoolExecutor*, a qual permite o agendamento de *timeouts*

7. Eliminar qualquer possibilidade de Bloqueio

Para uma escalabilidade máxima, deve ser eliminado o recurso a qualquer função bloqueante, nomeadamente, as chamadas de acesso ao sistema de ficheiros. Para isso poderão usar a classe *java.nio.channels.AsynchronousFileChannel*.

8. Conclusão

Este documento foca-se nos aspectos de concorrência do Projeto 1 e assume implicitamente que o ficheiro alvo do *backup* tem apenas um *chunk*. As sugestões antes apresentadas podem, de qualquer das formas, ser aplicadas ao *backup* de ficheiros com mais de um *chunk*, bem como à implementação dos outros subprotocolos.

Devem seguir uma abordagem incremental. No entanto, neste caso, vocês tem um "espaço de soluções" que é, pelo menos, bidimensional: numa das dimensões, está o nível de simultaneidade e escalabilidade pretendido para a vossa solução e, na outra dimensão, estão os subprotocolos (o número de *chunks* por arquivo adiciona, de certa forma, uma terceira dimensão).

A melhor abordagem será, provavelmente, uma do tipo "depth-first", na qual vocês primeiro implementam o *Chunk Backup Protocol* com o nível de simultaneidade que desejarem. Depois implementam os outros protocolos com o mesmo nível de simultaneidade. Devem, no entanto, gerir o vosso tempo: se demorarem muito para tratar da simultaneidade, depois talvez vos falte tempo para implementar os restantes protocolos.

9. Leitura Adicional

Existem certamente muitos recursos bibliográficos relativos ao assunto que aqui abordamos. Os seguintes tutoriais parecem-me, no entanto, apresentar um bom equilíbrio entre teoria e prática e, portanto, podem ser úteis ao desenho e implementação do vosso primeiro projeto de SDIS (em Inglês):

- Java Concurrency/Multithreading Tutorial – A tutorial discussing concurrency issues in the context of Java. You may wish to take a look at the Concurrency Models chapter, which discusses several concurrency designs more abstractly than these notes.
- Java Concurrency Utilities – A tutorial on the *java.util.concurrent* package, including *ThreadPoolExecutor* and *ScheduledExecutorService*, which is the interface provided by the class *ScheduledThreadPoolExecutor*.
- Java NIO Tutorial – A good tutorial on Java NIO, including the *java.nio.channels.AsynchronousFileChannel*.