SDIS Projeto 1

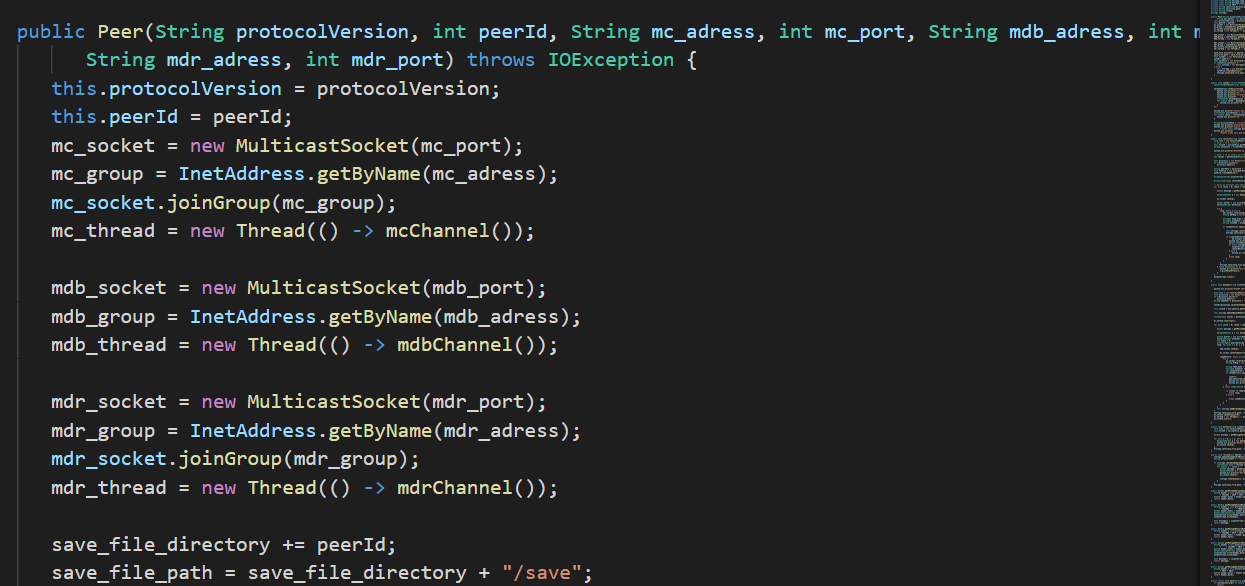
BackUp Service

Ana Macedo up201909572;Miguel castro up201909578

2020

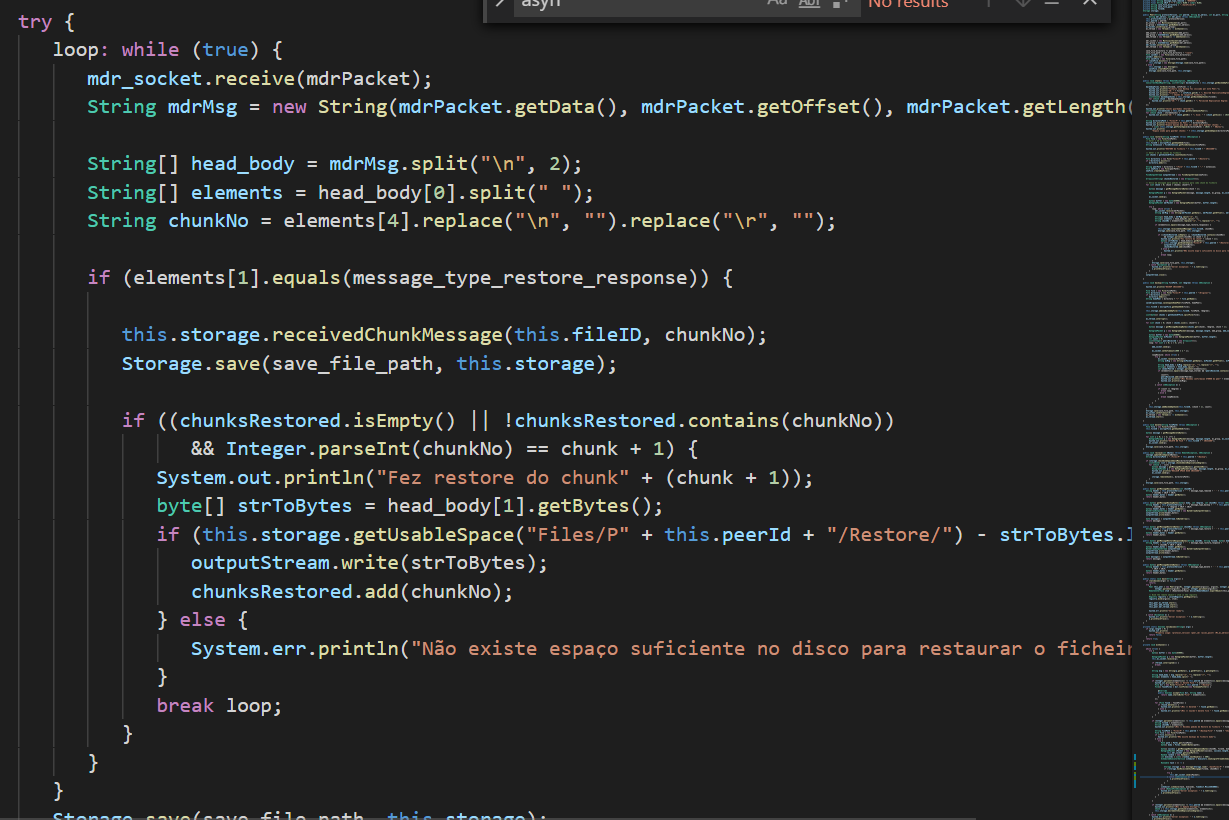
# Design de Concorrência

Ao mesmo tempo que se inicializam os *Peers*, no seu construtor, também se inicializam os canais MC, MDB e MDR, subscrevemos os grupos (com a porta e endereço dados como argumento na invocação do *peer)* e criamos as *threads,* uma por canal.



## One Thread per Multicast Channel, Many Protocol Instances at a Time

Usamos uma thread receptora por canal, e cada uma completa o processamento de uma mensagem, antes de iniciar o processamento da seguinte mensagem, como se vê na figura seguinte que representa a receção da mensagem CHUNK no protocolo Restore e o tratamento do corpo da mensagem recebido.

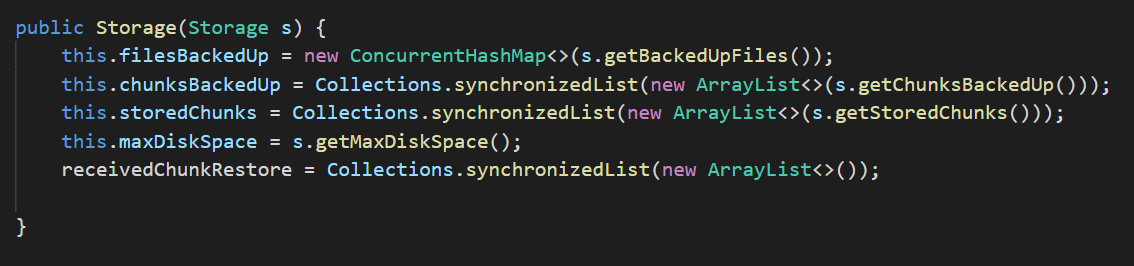


Assim, cada thread executa um loop infinito e em cada iteração recebe a mensagem no respetivo canal multicast e processa a mensagem recebida.

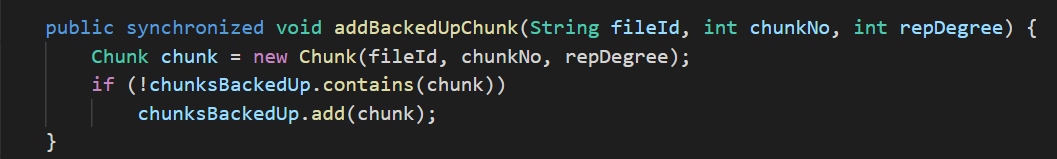
## Uso de listas/*Maps* sincronizados (thread-safe)

Na classe Storage várias threads podem aceder aos seus objetos e alterá-los ao mesmo tempo, não sendo o desejado. Para solucionar este problema é necessário o uso da classe thread safe, como a ConcurrentHashMaps e Collections.synchronizedList.

Na figura seguinte encontra-se inicializações de objetos das classes referidas.



Para além disso, os métodos onde se alteram esses objetos são declarados como *synchronized.*

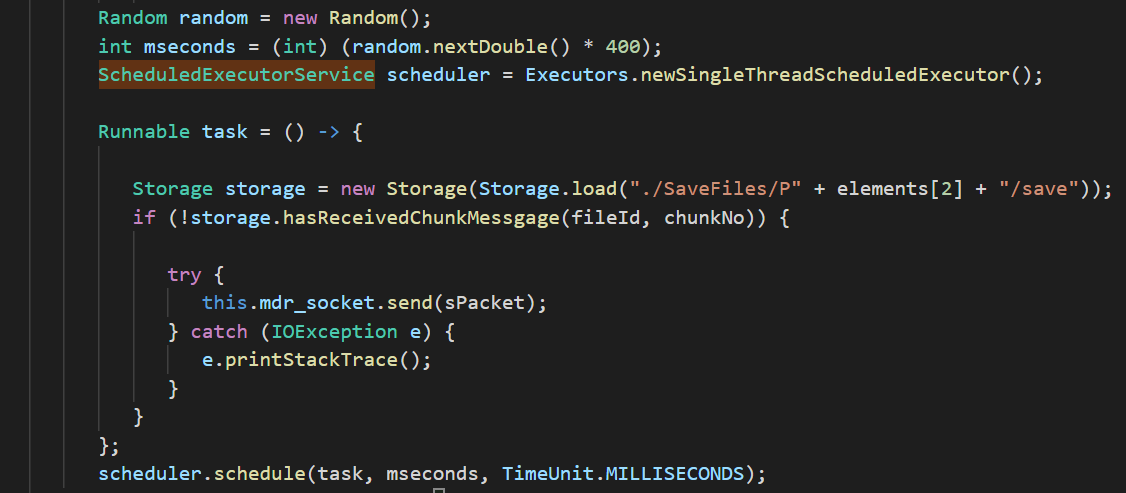


## Implementação escalável

Para vários subprotocolos, era necessário cada *Peer* esperar um tempo aleatório uniformemente distribuído entre 0 e 400 ms antes de enviar uma mensagem para o *initiator Peer* ou para executar o subprotocolo backup.

Para uma implementação escalável foi substituída a função *Thread.sleep()* pela classe *java.util.concurrent.ScheduledExecutorService.*

Na imagem encontra-se um exemplo de implementação, em que o *mseconds* são os milissegundos gerados aleatoriamente.



Neste exemplo, no subprotocolo restore, só se envia a mensagem CHUNK depois de um tempo aleatório.

A função Schedule permite que o que está dentro do bloco *task* seja executado depois dos milissegundos dados.

Este método é repetido no subprotocolo backup e reclaim, sendo as task() diferentes.

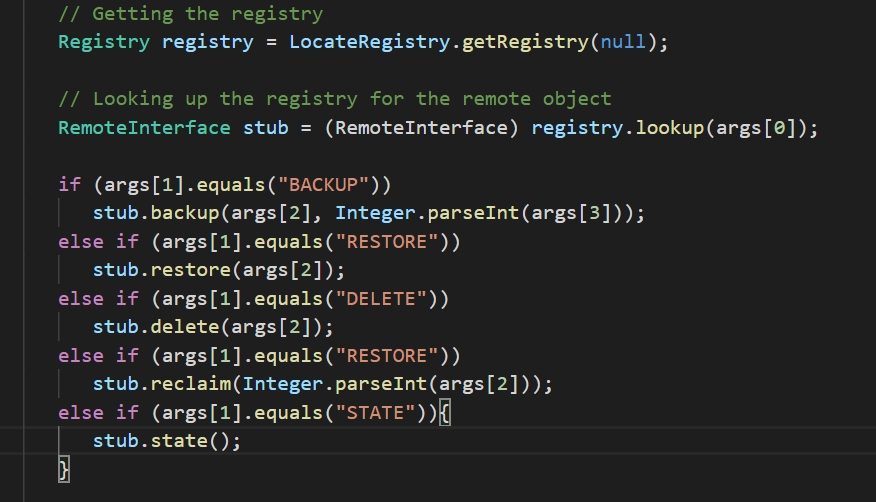
Em relação à classe java.nio.channels.AsynchronousFileChannel foi tentada implementar, mas sem sucesso.

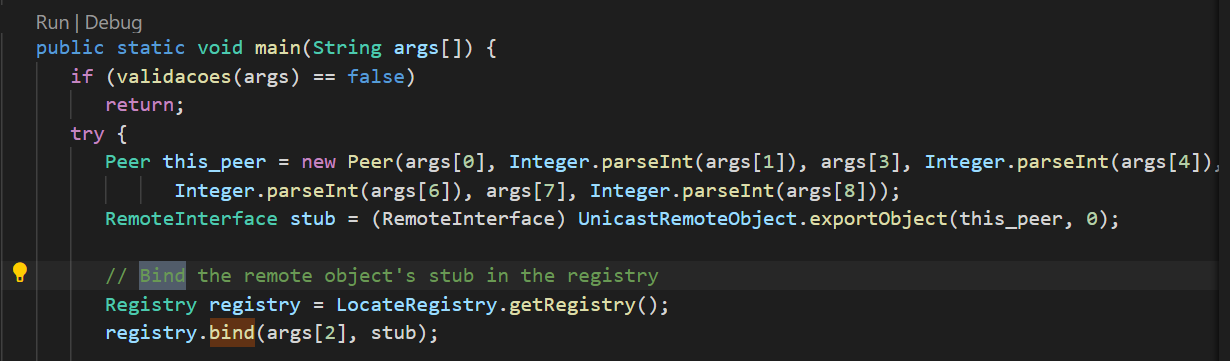
## Comunicação entre o TestApp e o Initiator Peer

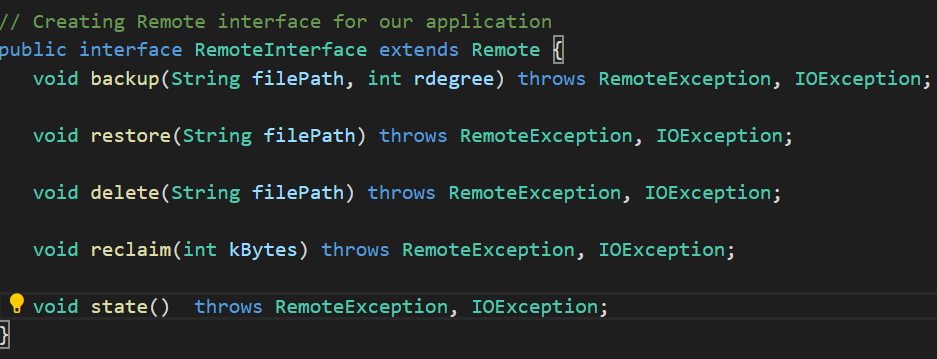
Para a comunicação entre o cliente e o *initiator peer* foi usado RMI.

O nome da referência remota usado no método lookUp() do cliente e no bind() *do initiator peer* é o *peer access point* dado no primeiro argumento, quando se invoca o cliente.

Nas figuras a baixo encontram-se blocos de código da implementação do RMI, TestApp, Peer e RemoteInterface respetivamente.







# Melhorias

Não foram implementadas nenhumas melhorias no projeto.