U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

SISTEMAS DISTRIBUÍDOS

RELATÓRIO DO PRIMEIRO PROJETO 2019/2020

Raul Viana up201208089@fe.up.pt
João Lemos ee10201@fe.up.pt

Introdução

Este relatório tem como objetivo contextualizar as opções tomadas durante o desenvolvimento do trabalho prático.

Apenas foi realizado melhoramento ao protocolo *BACKUP*. Este melhoramento teve como resultado uma melhoria na eficiência e na gestão da memória. O protocolo de recuperação de espaço foi apenas parcialmente implementado.

Foram ainda utilizados alguns pormenores que permitem a execução simultânea de subprotocolos e execução concorrente intra-subprotocolo.

Melhorias aos protocolos especificados

Apenas o protocolo *BACKUP* foi sujeito a melhoramento. No entanto, este melhoramento foi idealizado de forma a manter interoperabilidade e, desta forma, é desnecessário verificar o número da versão, pois o protocolo mantém todas as funcionalidades base e forma de funcionamento. É assim possível garantir a interoperabilidade com as mensagens descritas na especificação.

De forma a garantir o grau de replicação de cada *chunk* foram criadas várias estruturas de dados relativas aos ficheiros cujo *peer* fez uma cópia de segurança e aos *chunks* guardados a pedido de outros *peers*.

Cada pedido de *PUTCHUNK* leva a que o *peer* que o recebe comece por confirmar as suas *remoteOcurrences*. Esta tabela contém informação acerca do grau de replicação atual de cada *chunk.* Na eventualidade de o grau de replicação gravado já ter atingido o grau de replicação desejado o *chunk* é descartado e a escrita abortada.

Deste modo é possível ganhar alguma eficiência e memória, uma vez que não são processados e guardados *chunks* desnecessariamente.

Execução concorrente

Foram implementados diversos pormenores, que permitem à aplicação executar diversas tarefas simultaneamente e de uma forma concorrencial.

As estruturas de dados escolhidas para isso são de extrema importância, pois podem ou não ser adequadas a ambientes *multi-threading*. Assim foram escolhidas para algumas estruturas de dados implementações de *ConcurrentHashMap.* Isto porque esta implementação, num ambiente *multi-threading* é mais segura, escalável e permite atingir um desempenho excelente mesmo com um rácio de *threads* concorrenciais de leitura e escrita muito desequilibrado.

Foi criada uma *thread pool* através da instanciação de um objeto do tipo *ScheduleThreadPoolExecutor* que permite criar e iniciar *threads* de uma forma muito diversa. Por exemplo é possível simplesmente criar uma *thread*, criar uma *thread* que será iniciada passado determinado tempo, ou ainda criar uma *thread* que se repetirá com um intervalo regular de tempo.

```
Peer.getExecutor().schedule(
        new RepeatPutChunkMessage(message, 1, file.getID(), chunk.getID(), replication_degree), 1,
        TimeUnit.SECONDS);
```

Figura 1 - Exemplo de chamada de thread com atraso

Na classe **Peer** é criado um objeto para cada canal *multicast*, *MC_Channel, MDB_Channel* e *MDR_Channel,* que são instâncias da classe **MulticastCom.** Os objetos desta classe processam cada mensagem recebida criando uma *thread,* de forma a que seja possível receber várias mensagens ao mesmo tempo, sem que a aplicação bloqueie durante o processamento de cada uma delas.

```
while(true){

    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
    receiveSocket.receive(packet);
    byte[] receivedMessage = Arrays.copyOf(buffer, packet.getLength());
    Peer.getExecutor().execute(new IncomingManager(receivedMessage));
}
```

Figura 2 - Exemplo da gestão das mensagens recebidas

Para além de estruturas de dados adequadas ao *multi-threading* foi ainda utilizado o *synchronized.* A utilização de *threads* concorrenciais pode levantar alguns problemas de acesso assíncrono aos dados ou *deadlocks*. Ao utilizar o *synchronized* em blocos de código sensíveis tira-se partido da sincronização *built-in* do Java, limitando assim a probabilidade de acontecerem estes problemas inerentes ao *multi-threading*. Tentou-se limitar o tamanho dos blocos marcados com *synchronized*, para que a eficiência não fosse comprometida, mas devido à grande quantidade de *threads* e de

estruturas de dados com acesso concorrencial esse objetivo não foi plenamente alcançado.

De forma a manter o estado de cada **Peer** entre cada execução foi implementada uma forma de guardar este mesmo estado em disco. Ao encerrar a aplicação, esta executa um método que guarda o seu estado interno para um ficheiro, e ao iniciar carrega esse mesmo ficheiro. Esta funcionalidade foi obtida através da utilização do mecanismo de serialização do Java. Este mecanismo guarda um objeto numa sequência de *bytes*. O mecanismo inverso carrega o objeto para a memória, ficando este com as informações que tinha anteriormente ao encerramento.