DESCRIÇÃO DA ARQUITETURA CONCORRENTE

Cada peer tem uma arquitetura completamente concorrente ao nível dos protocolos, concorrência essa atingida através de multithreading. A descrição desta arquitectura será apresentada abaixo. Visto que em Java uma thread não é mais do que uma classe que implementa a interface **Runnable**, todas as threads nesta descrição serão referidas pelo nome da classe que as implementa.

A thread principal, **Peer**, inicializa quatro dispatchers, cada um deles uma thread: três dispatchers para cada um dos 3 canais multicast e um dispatcher para pedidos do cliente via RMI. Após criar e iniciar essas 4 threads, a thread Peer limita-se a guardar o estado interno do Peer em memória não volátil em intervalos de 2 segundos e recorrendo ao mecanismo de serialização do Java.

As três dispatcher threads para os canais de multicast, **DispatcherMC**, **DispatcherMDB** e **DispatcherMDR**, funcionam de forma exatamente igual: esperam que um datagrama chegue à socket, lêm o datagrama e olham para o cabeçalho da mensagem de modo a saber qual o tipo de worker thread (denominadas de handlers neste contexto) que deve processar a mensagem. O handler é seguidamente executado recorrendo a um **ThreadPoolExecutor**, que limita o número de handlers em execução (capacidade base para 200 handlers e máxima para 400). Logo após mandar executar o handler, o dispatcher volta imediatamente a ficar à escuta de datagramas.

A dispatcher thread para RMI, **DispatcherRMI**, funciona de forma semelhante: para cada pedido recebido, inicia um handler e fica logo pronto para receber outro pedido. Só no caso de um pedido de RESTORE precisa de aguardar que o handler termine de modo a retornar o ficheiro reassemblado para o cliente. O pedido de STATE não necessita de handler, visto ter um retorno imediato com um overhead desprezável.

Existe um handler diferente para cada mensagem. Para as mensagens recebidas pelo DispatcherMC, handlers do tipo **StoredHandler**, **GetchunkHandler**, **DeleteHandler**, **RemovedHandler** e **OnlineHandler** são criados, sendo que a mensagem que visam processar corresponde à primeira parte dos seus nomes (a mensagem ONLINE faz parte de um enhancement). O DispatcherMDB lança apenas handlers do tipo **PutchunkHandler** e o DispatcherMDR handlers do tipo **ChunkHandler**. Por fim, o DispatcherRMI lança handlers do tipo **BackupHandler**, **DeletionHandler**, **ReclamationHandler** e **RestoreHandler**, sendo que equivalem cada um aos pedidos que um cliente pode fazer ao peer.

Estes handlers executam a lógica do protocolo inerente à mensagem que lhes é subjacente e, regra geral, não iniciam threads dentro dessa lógica. As únicas exceções ocorrem no RestoreHandler, que lança threads ChunkHandler, e no OnlineHandler, que lança threads do tipo DeletionHandler. Ambas estas exceções fazem parte de enhancements descritos posteriormente neste relatório.

De modo a criar uma estrutura de dados comum a todas as threads, foi criado um singleton chamado **Manager** que possui informação estática para acesso facilitado e estruturas dinâmicas para o armazenamento de informação inerente à lógica dos protocolos (chunks guardados, replication degrees, mensagens delete enviadas, etc). Toda esta informação é thread-safe, tendo sido criadas classes usado métodos assinados como **synchronized** de modo a evitar estas situações, assim como **ConcurrentHashMap**s da biblioteca standard do Java como atributos dessas classes.

As estruturas existentes no Manager são:

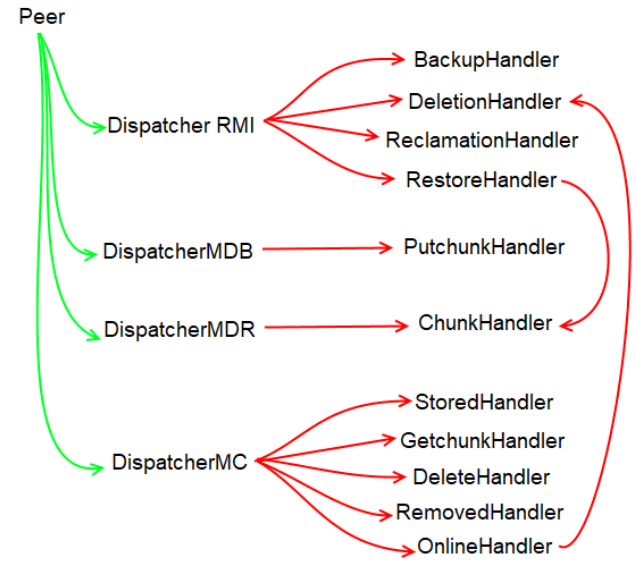
- **BackupManager**: guarda a informação sobre um pedido de BACKUP no initiator peer desse pedido. Serve para verificar se os chunks propagaram com um replication degree apropriado, assim como verificar quais os ficheiros que o peer tratou.

- **ChunkManager**: guarda e gere objetos **Chunk**, que representam um chunk que pode ou não estar guardado nesta máquina (ver enhancement do BACKUP), assim como o replication degree estimado para esse chunk.

- **RestoreManager**: guarda temporariamente os chunks que estão a ser recuperados de modo a reconstituir um ficheiro, e faz essa própria reconstituição.

- **MessageRegister**: guarda um registo de quando uma mensagem chega, visto que a simples chegada de certas mensagens pode ter influência sobre certos subprotocolos não diretamente relacionados com essas mensagens. Existem duas instâncias, uma para mensagens PUTCHUNK e outra para mensagens CHUNK.

A sequência de criação de threads pode, então, ser sumariada no seguinte diagrama:



Setas verdes: lança apenas uma instância
Setas vermelhas: lança múltiplas instâncias

ENHANCEMENT – Subprotocolo RESTORE

O problema subjacente ao subprotocolo RESTORE especificado prendia-se com o facto de se sobrecarregar o canal de multicast com várias mensagens CHUNK repetidas quando de facto não são necessárias tantas para reconstituir um ficheiro. Como tal, foi criada uma nova versão do protocolo, **1.1**, com vista a realizar esta transmissão usando TCP. Para tal efeito, foi adicionado um novo header à mensagem GETCHUNK (implementada pela classe **MessageGetchunkEnhanced**):

```
GETCHUNK 1.1 <SenderId> <FileId> <ChunkNo> <CRLF>
          <TCP server port> <TCP server IP> <CRLF><CRLF>
```

Esta mensagem, como é evidente, transporta a informação referente a uma socket TCP estabelecida no initiator peer (o peer em que está a realizar o RESTORE). Um peer que receba esta mensagem irá conectar-se a essa socket através de uma conexão TCP usando a informação recebida, e enviará a mensagem CHUNK através dessa conexão. O initiator peer estará, por sua vez, a aceitar um pedido de conexão TCP, a ler a stream de informação até acabar (ou seja, até ao fim da mensagem CHUNK) e a iniciar uma thread **ChunkHandler** para processar essa mensagem. Depois de lançar a thread, fecha a conexão, aceita um outro pedido de conexão e repete o processo até não haverem mais conexões para aceitar/mensagens CHUNK para receber.

O protocolo é totalmente compatível com peers que não suportam esta versão do protocolo: esses peers mandarão as mensagens CHUNK pelo canal de multicast e essas mensagens serão processadas normalmente. O processo de reconstituição é completamente agnóstico ao método de transporte dos chunks, ou seja, um ficheiro reassemblado poderá ter chunks provenientes de ambos os meios de transmissão (dependendo, claro, do número de peers ativos que suportam ou não o protocolo melhorado).

ENHANCEMENT – Subprotocolo BACKUP

O problema no protocolo de BACKUP prende-se com o facto de os peers poderem armazenar mais cópias dos chunks do que o replication degree desejado, o que pode causar problemas de espaço. Como tal, foi necessário arranjar uma solução que tentasse garantir que os peers não armazenassem mais chunks do que aqueles estritamente necessários. Portanto, o problema aqui prende-se em saber exatamente quando é que este limite é atingido. Ora, a replication count é aumentada sempre que um peer recebe

uma mensagem de STORED, que lhe indica que outro peer armazenou a mensagem. Da forma como o protocolo original está especificado esta informação é apenas útil ao intiator peer, mas neste enhancement foi reutilizada de modo a fazer com que os vários peers consigam perceber o estado atual da replication count geral. Para tal efeito, sempre que uma mensagem PUTCHUNK é recebida, o peer não guarda logo o chunk em disco; em vez disso, toma nota da chegada do chunk e estabelece uma replication count de 0 para esse chunk. Depois, durante a espera aleatória de 0-400ms presente no protocolo original todas as mensagens STORED recebidas e referentes a esse chunk fazem com que a replication count desse chunk aumente uma unidade por mensagem. Quando o tempo de espera termina, se a replication count obtida pela leitura de mensagens STORED for maior do que a replication count desejada para esse chunk, toda a informação sobre o chunk é removida do peer e o chunk não é guardado em disco. Se for menor, a informação do chunk permanece no peer e o conteúdo do chunk é finalmente guardado em disco, procedendo-se de seguida à transmissão de uma mensagem STORED.

Este método não garante que absolutamente todos os chunks tenham exatamente o número mínimo de cópias, visto que mensagens STORED podem chegar quando o peer está a escrever para o disco ou a transmitir a mensagem, mas na maior parte dos casos é garantido que esse número é mínimo, sendo que as situações em que é superior, quando acontecem, não diferem muito mais do que uma ou duas unidades.

ENHANCEMENT – Subprotocolo DELETE

O problema subjacente ao protocolo de DELETE original tem a ver com o facto de um peer que armazene chunks que precisam de ser apagados não estar online no momento em que a mensagem DELETE é enviada. Ora, dada a natureza agnóstica do sistema um peer nunca sabe quandos peers é que estão online nem quando estes vão online ou offline. Como tal, foi criada uma mensagem adicional, ONLINE, que indica que um peer ficou online. Esta mensagem, como seria de esperar, é enviada pelo próprio peer mal este fique online, sendo que essa transmissão ocorre na classe **Peer**, logo após criar os dispatchers para os datagramas multicast. Esta mensagem é enviada no canal MC, pertence à versão **1.1** do protocolo e tem o seguinte formato:

```
ONLINE 1.1 <SenderId> <CRLF><CRLF>
```

Como é possível observar, esta mensagem não carrega qualquer tipo de informação relevante; serve apenas para sinalizar que um novo peer ficou online.

Agora que é possível saber quando um peer fica online, é necessário enviar-lhe as mensagens DELETE que ele perdeu enquanto esteve offline. Para tal, cada initiator peer do subprotocolo DELETE guarda informação sobre as mensagens DELETE que enviou, e sempre que um peer fica online os initiator peers iniciam novamente o subprotocolo DELETE para todos os pedidos que armazenaram. Os peers que estavam online no momento em que estas mensagens foram originalmente enviadas não vão fazer nada, mas o peer que ficou recentemente online vai processá-las e remover os chunks referentes à mensagem, caso os tenha. O registo das mensagens DELETE nos initiator peers é, em princípio, permanente, sendo que são removidas quando o initiator peer recebe ou um pedido de BACKUP do mesmo ficheiro por parte do cliente ou uma mensagem PUTCHUNK proveniente de outro peer (o que indica que um cliente pediu a outro peer para iniciar o subprotocolo BACKUP para esse ficheiro). As mensagens de PUTCHUNK podem também ter origem no subprotocolo RECLAIM, mas visto que supostamente não existe em nenhum peer chunks desse ficheiro para remover de modo a libertar espaço, essa situação nunca acontecerá.