Implementação concorrente

Para garantir execução concorrente de protocolos, o sistema segue uma estrutura conceptualmente baseada no modelo de concorrência *Assembly Line* (também conhecido por *Event-Driven*), onde múltiplos *Worker Threads* dividem entre si as operações pedidas de forma sequencial.

No geral, pode-se dividir a execução de um protocolo em cinco estruturas, que possuem independentemente os recursos necessários para execução concorrente das operações que lhes são pedidas:

- *Initiator:* caso o *peer* seja initiator do serviço, um *worker* é iniciado onde é executada a lógica do protocolo pedido.
- *Receiver:* estrutura onde são feitas as leituras das mensagens recebidas nos canais *multicast*, para serem interpretadas e processadas.
- *Dispatcher:* estrutura onde as mensagens (eventos) recebidas pelo *peer* são interpretadas, e de seguida encaminhadas para processamento por um novo *worker*, em função do evento.
- *PeerController:* estrutura de processamento dos eventos propriamente ditos, onde é mantida a lógica associada às instâncias dos vários protocolos. Os *workers* de processamento utilizam essa informação de forma concorrente.
- FileSystem: estrutura onde são executadas as operações de I/O associadas a uma instância do protocolo. Os *workers* associados a esta estrutura acedem de forma concorrente aos ficheiros guardados em memória, através de funções da biblioteca Java NIO e, em alguns casos, métodos assinalados como *synchronized.*

É de destacar a presença geral de *Thread Pools* com possibilidade de *scheduling* nas várias fases descritas acima, de forma a garantir escalamento e reutilização de recursos na execução. Da mesma forma, a utilização estruturas de dados concorrentes, nomeadamente da biblioteca `java.util.concurrent`, permite manter os dados associados às instâncias dos protocolos de forma *thread-safe*.

Desta forma, o sistema tem a possibilidade de servir as várias fases dos pedidos de protocolo de forma independente e concorrente, embora com alguma informação comum entre eles.

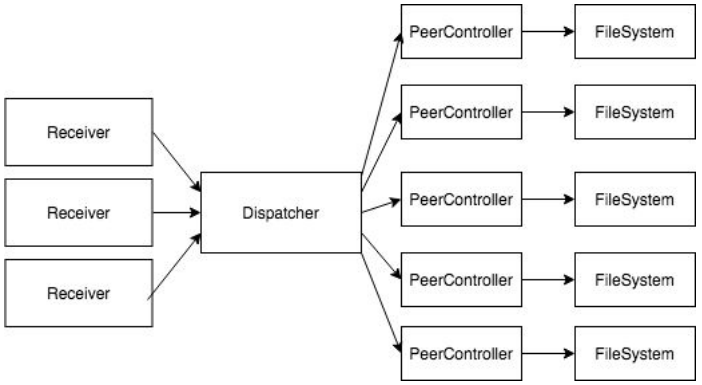

Figura 1: Modelo conceptual dos *workers* do sistema, desde a receção de uma mensagem à possível execução de operações I/O.

Melhoria backup

Para este protocolo, o objetivo era garantir que, dentro do possível, o grau de replicação de um ficheiro fosse respeitado. No protocolo original, o initiator peer, após 1 segundo de espera, apenas verifica se o número de respostas STORED é maior ou igual ao grau desejado. Neste tempo, não será incomum haver mais respostas do que o necessário para satisfazer o pedido (dada a existência de peers suficientes). Assim, gasta-se memória de forma desnecessária muito facilmente.

A nossa melhoria consiste em realizar uma escuta ativa do lado dos non-initiator peers: antes de processar o pedido de PUTCHUNK, é feita uma escuta durante um valor uniformemente aleatório até 0.75 segundos do canal de controlo por mensagens STORED relativas ao chunk em questão. Durante esta espera são registadas as respostas enviadas e, no fim, verificado o grau de replicação relativamente ao desejado: continuando a estar aquém, o peer guarda-o e envia também uma resposta STORED. Contudo, já estando o grau de replicação satisfeito, ignora a mensagem.

Desta forma, garante-se que os chunks de um ficheiro serão guardados com o grau de replicação desejado (estando reunidas as condições necessárias em termos de peers online), evitando o uso desnecessário de memória. Não havendo novas mensagens nem mudanças significativas ao nível do processamento, esta melhoria é compatível com a versão original do protocolo.

Melhoria restore

Para este protocolo, o objetivo era garantir que os canais multicast não fossem sobrecarregados com informação desnecessária, nomeadamente ao nível de envio de chunks para fazer restore de um ficheiro previamente guardado na rede. No protocolo original, as mensagens CHUNK, que contêm o conteúdo dos chunks, são enviadas por um canal multicast, no entanto a informação interessa a um único peer (o que desencadeou o protocolo).

A nossa melhoria consiste em aproveitar informação conhecida a priori entre todos os *peers*, mais concretamente o porto do canal MDR. Através deste porto é inicializado um *socket* através do qual serão estabelecidas as ligações necessárias para o envio dos chunks. Quando um non initiator peer recebe uma mensagem GETCHUNK e contém o chunk pedido, este extrai o endereço do *sender* da mensagem (o *initiator peer)* e, juntando o valor do porto, abre um *socket* nessa localização. É estabelecida, então, uma ligação TCP entre os dois peers e é feito o envio do conteúdo do chunk em questão. Este procedimento repete-se tantas vezes quanto necessário para o envio de todos os chunks do ficheiro. De forma a poupar recursos, os *sockets* entre os mesmo *peers* são aproveitados, caso sejam enviados múltiplos chunks entre os mesmos.

Paralelamente é feito o envio da mensagem CHUNK para o canal de restore, excluindo o conteúdo do chunk (body) dessa mensagem (contrariamente ao protocolo original). Visto que, para os outros *non initiator peers*, o único interesse de receber a mensagem é saber que aquele chunk já foi enviado (para evitar *flood* de informação redundante), esta ‘separação’ não tem influência no comportamento dos peers a correr o protocolo original, sendo, então, compatível.

Melhoria delete

Para este protocolo, o objetivo era garantir que os ficheiros para os quais foi pedido DELETE foram efetivamente apagados, mesmo quando um peer que tivesse chunks do ficheiro não estivesse a correr durante o pedido do protocolo.

A nossa melhoria consiste em guardar a informação de quais peers guardam pelo menos um chunk dos ficheiros (através das mensagens STORED), e quais ficheiros deveriam estar apagados no sistema. Com a introdução de uma mensagem extra HELLO, enviada quando um peer é iniciado, utiliza-se a informação guardada até ao momento para definir se é necessário iniciar novo pedido de DELETE para algum ficheiro.

Resumidamente, quando é recebida uma mensagem HELLO de um peer, verifica-se se o mesmo tem guardado algum chunk que deve estar apagado, e se sim, inicia-se o protocolo DELETE para o fileID desse chunk. Os peers que efetivamente apagaram os chunks do ficheiro na altura do primeiro pedido, simplesmente ignoram a mensagem. Assim a melhoria é também interoperável.

Finalmente, para garantir que é possível múltiplos backups e deletes de um ficheiro, após a receção de uma mensagem PUTCHUNK, é retirada a eventual ocorrência do ficheiro na listagem de ficheiros apagados.

António Almeida, up201505836
João Damas, up201504088