

Distributed Systems

*Project Report*

Mestrado Integrado em Engenharia Informática e Computação

Francisco Friande up201508213@fe.up.pt

Luís Mendes up2016@fe.up.pt

*Introduction*

This report aims to expose the strategies used in the project to manage the protocol concurrency issues, as well as to explain the only enhancement implemented: backup replication degree ensuring enhancement.

*Backup Protocol*

Given that the *“scheme [backup] can deplete the backup space rather rapidly, and cause too much activity on the nodes once that space is full.”,* we figured this could be a major problem in storage managing and overall performance of the project. To solve this problem, we had to come up with a solution that involves individual peer’s storage system handling and some asseveration on synchronization matters.

Thus, we implemented the following architecture: each Peer will manage its own chunks storage (class Storage), using the following data structures.

    private *ArrayList*<FileContent> files;

    private *ArrayList*<Chunk> storedChunks;

    private *ArrayList*<Chunk> restoredChunks;

    private *HashMap*<String, Integer> blackListedChunks;

    private *ConcurrentHashMap*<String, Integer> chunkOccurences;

    private *int* space;

In it, we will explain specifically the *chunkOccurences* ConcurrentHashMap. The aforementioned java native data structure manages concurrency of different threads trying to access the same HashMap, allowing only one thread at a time to do so. Because it provides thread-safety and memory-consistent atomic operations, it’s ideal for our final purpose.

To achieve it, every time a PUTCHUNK message results in an actual chunk save, Peer’s local *chunkOccurences* int value for the key (file id name + “/” + chunk id) is increased; following a STORED message being sent, which will tell the other peers to update their *chunkOccurences* too. With this mechanism, all Peers have the information they need not to store chunks beyond the desired number.

With a simple verification in the PUTCHUNK handling class, of how many times has the chunk been saved and if it has achieved the desired replication degree or not, the likelihood of storing more chunks than the degree is very low, also due to the random delay [0-400ms] to the actual storing of the chunk (scheduling the PUTCHUNK handling class thread).

*Protocol Thread Concurency*

Sim, nós recorremos a métodos sincronized do java. Em que cada peer iniciador irá executar a função correspondente ao que foi especificado via a interface RMI na class peer (seja essa instrulão backup, delete, restore ..) e depois, enviará mensagens multicast através de "canais" que no fundo são ainda outras threads (objetos runnable). Haverá, então, depois várias worker threads que representam os peers restantes (para além do iniciador) que será a HandleMessage, essa encarrega-se de reagir às mensagens que "vão sair" do outro lado do canal, conforme é PUTCHUNK, REMOVED, etc ..

ah, e está tudo na mesma threadpool, sendo uma ScheduledThreadPool, podendo nós agendar as execuções das ações (que dá jeito pq muitas vezes temos aquele delay)

There were many factos that we took into account to implement a reliable architecture that allows simultaneously running threads.

As mentioned before, ConcurrentHashMap is a way to ensure safe thread concurrency, revealing a remarkable performance in access and modification management for multiple threads.