Concurrency implementation

The approach chosen to implement subprotocols concurrency was the "Processing of Different Messages Received on the Same Channel at the Same Time". This is used to execute all the subprotocols.

The major idea behind this is to have an **ExecutorService** implementing a **ThreadPool**. When initializing the **SubprotocolManager**, responsible for receiving the messages from the multicast channels and invoking the corresponding functions/subprotocols, the **ThreadPool** set with a maximum of 5 threads.

This means that at the same time, there will be at most 5 threads being executed.

When a message is received it is added to a **ConcurrentLinkedQueue**, responsible for storing the received messages.

Once a thread becomes available the executor will make the request contained in the first message present in the queue.

To complement this, each peer has a **ConcurrentHashMap** in its fileHandler. This map will contain the stored chunks as well as information regarding the chunks stored in other peers. Being a **ConcurrentHashMap** versus a **HashMap**, makes it thread-safe, meaning that it can only be accessed by a single thread at a time.

The most part of the concurrency implementation is in the **SubprotocolManager** file and some methods present in other files are synchronized. Having synchronized methods prevents thread interference as well as memory consistency errors.

Backup Enhancement

Problem

This scheme can deplete the backup space rather rapidly and cause too much activity on the nodes once that space is full. Can you think of an alternative scheme that ensures the desired replication degree, avoids these problems, and, nevertheless, can interoperate with peers that execute the chunk backup protocol described above?

Enhancement

In our service when a Peer receives a STORED message (sent by Peers that have stored a chunk) it stores the PeerID of the Peer that sent that message and increases the replication degree of that chunk. When a Peer receives a PUTCHUNK message (sent by the initiator Peer) it waits a random time between 0 and 400ms and then checks if the current replication degree of the chunk to store is already greater than the replication degree that it is supposed to have. In the case that replication degree is not greater, the chunk is stored in the current Peer.