14th April 2019

Faculdade de Engenharia da Universidade do Porto

Distributed Backup Service

Distributed Systems

Class 1 - Group 14

João Fernando da Costa Meireles Barbosa – up201604156

Concurrency Design

The Backup Service achieves concurrency through the use of multiple threads. The design hereby presented, employed in the service, has the following characteristics:

- Concurrent execution of operations (Backup, Restore, Delete, Reclaim and State);
- Parallelization of the Backup and Restore protocols;
- Concurrent processing of messages received through the multicast channels;

Operation Concurrency

The implemented Backup Service allows for multiple operations to be executed at the same time. In order for this to be achieved, two problems arise.

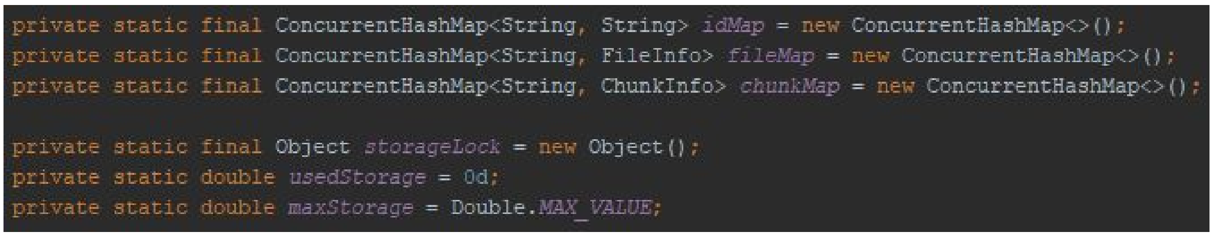
Firstly, the software must be capable of running multiple instances of operations at once. This issue is effortlessly solved through the use of RMI to establish the Client to Peer connection. Remote methods can be ran in parallel, since a new thread is started if needed for the execution of the invoked method.

The second and most troublesome issue is ensuring synchronization of the threads responsible for each operation when accessing common resources (metadata of backed up files and stored chunks, as well as the chunks themselves). This issue was solved through the use of java.util.concurrent.ConcurrentHashMap<K,V>, a thread-safe implementation of an hashtable, and explicit locks (synchronized blocks).

The files and chunks of a Peer are mostly managed through the ***storage.StorageManager*** class. This classes makes use of four *ConcurrentHashMap*s to manage this data:

- Mapping between a file's path and its generated fileId - ***StorageManager.idMap***;
- Mapping between a fileId and its relevant data (Initiator Peer) - ***StorageManager.fileMap;***
- Mapping between a stored chunk (fileId, chunkNo) and its relevant data - ***StorageManager.chunkMap***;
- (Backup Improvement) Mapping between a chunk (fileId, chunkNo) and its perceived replication **StorageManager.replicationMap**;

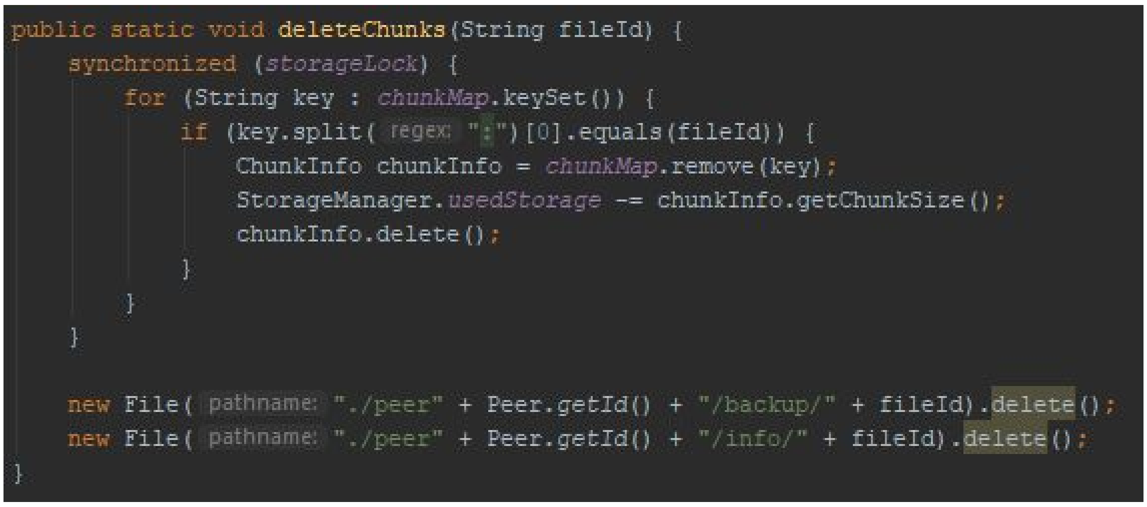
The ***StorageManager*** also keeps track of the used and maximum storage space (the latter defined by the Reclaim operation). These variables are synchronized using a simple object as a lock.



```java
private static final ConcurrentHashMap<String, String> idMap = new ConcurrentHashMap<>();
private static final ConcurrentHashMap<String, FileInfo> fileMap = new ConcurrentHashMap<>();
private static final ConcurrentHashMap<String, ChunkInfo> chunkMap = new ConcurrentHashMap<>();

private static final Object storageLock = new Object();
private static double usedStorage = 0d;
private static double maxStorage = Double.MAX_VALUE;
```

Declaration of the ***StorageManager*** attributes (hash tables, storage space and its lock)



```java
public static void deleteChunks(String fileId) {
    synchronized (storageLock) {
        for (String key : chunkMap.keySet()) {
            if (key.split( regex: ":")[0].equals(fileId)) {
                ChunkInfo chunkInfo = chunkMap.remove(key);
                StorageManager.usedStorage -= chunkInfo.getChunkSize();
                chunkInfo.delete();
            }
        }
    }

    new File( pathname: "./peer" + Peer.getId() + "/backup/" + fileId).delete();
    new File( pathname: "./peer" + Peer.getId() + "/info/" + fileId).delete();
}
```

StorageManager method responsible for deleting all chunks of a file, showing the use of the lately mentioned resources

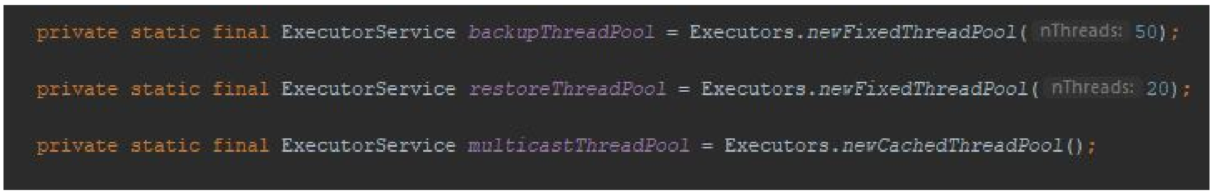
Synchronized blocks are also used to synchronize file access. Since multiple operations may require accessing a file containing a chunk's data or metadata, the ***storage.ChunkInfo*** class uses this method to synchronize threads accessing its files, using the file itself as a lock.

Backup, Restore and Multicast Concurrency

The Backup Service optimizes the Backup and the Restore operations by executing the corresponding chunk protocols in parallel. Similarly, received messages from the multicast channels are processed in parallel. To achieve this, each peer has three thread pools, two meant for the processing of multiple chunk protocols for each operation and one for the processing of multiple messages received by the multicast channel threads.

The *ThreadPoolExecutor* is a customizable class which creates and manages a thread pool while also providing a service for submitting tasks to be executed. In case this level of customization is not necessary (pool bounds, thread construction and lifetime, task queue, rejection handler, etc…), the *Executors* class provides factories for common preset thread pools. Among these include a fixed size thread pool, an unbounded thread pool with automatic thread reclamation, and a single background thread.

The main decision is determining what kinds of thread pools should be used for the backup, restore and multicast. The multicast channels are subject to a high load variation, since this load depends on the operations being executed in all of the network's peers. For this reason, the implemented design uses an unbounded thread pool for the multicast message processing, while the backup and restore operations use a fixed size thread pool each to limit their maximum throughput (to avoid having too much load on the network).

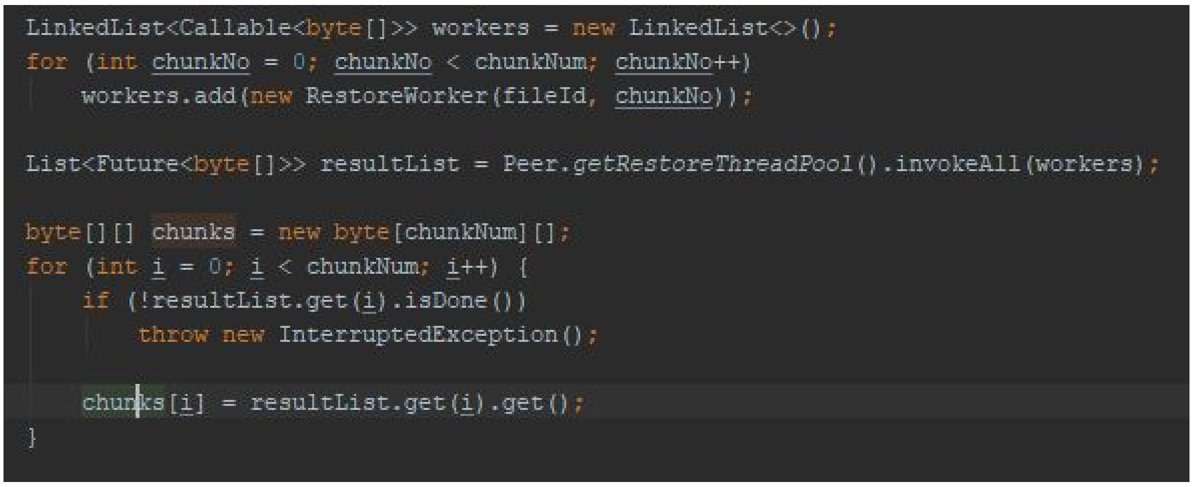


```java
private static final ExecutorService backupThreadPool = Executors.newFixedThreadPool( nThreads: 50);

private static final ExecutorService restoreThreadPool = Executors.newFixedThreadPool( nThreads: 20);

private static final ExecutorService multicastThreadPool = Executors.newCachedThreadPool();
```

Declaration of the three thread pools on the ***peer.Peer*** class



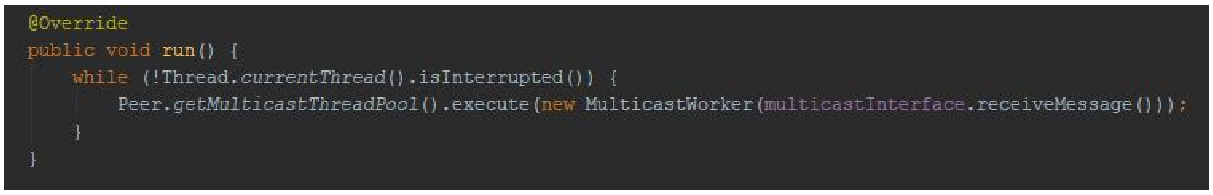
```java
LinkedList<Callable<byte[]>> workers = new LinkedList<>();
for (int chunkNo = 0; chunkNo < chunkNum; chunkNo++)
    workers.add(new RestoreWorker(fileId, chunkNo));

List<Future<byte[]>> resultList = Peer.getRestoreThreadPool().invokeAll(workers);

byte[][] chunks = new byte[chunkNum][];
for (int i = 0; i < chunkNum; i++) {
    if (!resultList.get(i).isDone())
        throw new InterruptedException();

    chunks[i] = resultList.get(i).get();
}
```

Concurrency of the restore operation (***peer.Service::restore***). A task is created for each chunk, then they're assigned to the thread pool and, after all the tasks are finished, each of their results are processed. A similar process is used for the backup operation.



```java
@Override
public void run() {
    while (!Thread.currentThread().isInterrupted()) {
        Peer.getMulticastThreadPool().execute(new MulticastWorker(multicastInterface.receiveMessage()));
    }
}
```

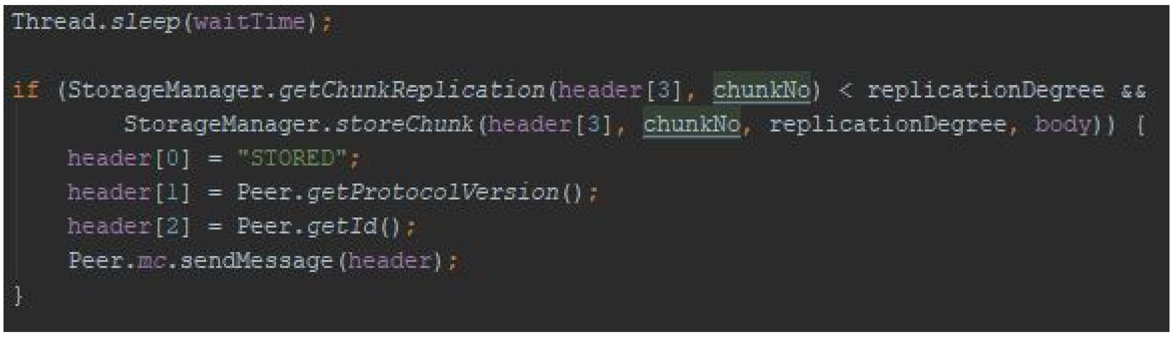
Body of the ***multicast.MulticastThread::run*** method. This class, created for each multicast channel, has the sole responsibility of receiving messages and dispatch them to a worker thread to process them. This assignment is managed by the multicast thread pool.

Backup Improvement

The vanilla chunk backup subprotocol stores a chunk regardless of its current replication. Depending on the number of active peers and the desired replication degree, this can cause the chunk to be replicated much more than the desired amount, resulting in wasted storage and unnecessary load on the network.

This problem can be avoided by simply changing how a peer reacts to a *PUTCHUNK* message. Following this approach, this improvement doesn't affect the initiator peer nor requires extra messages, so it is completely interoperable with peers running the vanilla version of the protocol, regardless of them being an initiator or receiver peer.

As stated, in the vanilla protocol, a peer receiving a *PUTCHUNK* message immediately stores that chunk regardless of its replication. If the processing of the message occurs at different times for each of the receiver peers, it is possible to first check the perceived replication of that chunk before inserting it, therefore solving the issues mentioned at the beginning. The vanilla protocol already does this, albeit only for the initiator peer, by using a random timer for each peer before it sends its response. By moving this random timer right after receiving a *PUTCHUNK*, it becomes possible for most of all other peers to keep track of the replication.



```
Thread.sleep(waitTime);

if (StorageManager.getChunkReplication(header[3], chunkNo) < replicationDegree &&
        StorageManager.storeChunk(header[3], chunkNo, replicationDegree, body)) {
    header[0] = "STORED";
    header[1] = Peer.getProtocolVersion();
    header[2] = Peer.getId();
    Peer.mc.sendMessage(header);
}
```

Enhanced algorithm used upon receiving a PUTCHUNK message. Essentially, the timer is moved to the start and the perceived replication is checked before storing the chunk.

The remaining issue is how to keep track of the replication. This replication is already tracked by the peers in the vanilla protocol, but only after the chunk's been stored and this replication is stored on non-volatile memory. The implemented solution keeps a separate temporary cache of the replication of non stored chunks, using an hash table for the mapping (***StorageManager.replicationMap***). This method keeps the original mechanism used to store chunk metadata untouched and respects the concurrency features aforementioned.