Concurrency Execution of Protocols

In our project, we chose to use RMI in the communication between the test application and the peer. With RMI, a new thread is created to handle every request that reaches a peer. So, with different threads to handle each request, it is possible for a peer to be executing multiple protocols at the same time.

RMI only creates different threads for each protocol, so we had to worry about each protocol being executed not to conflict with others. For each protocol, we will describe our implementation in terms of its concurrency.

Also, for each channel where information flows, we had to be careful so that information would not conflict. For that, for each channel a thread is created to listen to every message in that thread and whenever a message is received, a new thread is created to handle that message. Creating and Terminating threads has some overhead, so we used a *ThreadPoolExecutor* to execute every thread needed, being it in the handling of protocol execution request from test application or messages received on any channel.

```
executor = (ThreadPoolExecutor) Executors.newScheduledThreadPool(100);
executor.execute(new MCThread(this));
executor.execute(new MDBThread(this));
executor.execute(new MDRThread(this));
```

Creating *ThreadPoolExecutor* and executing the 3 channel listener threads

```
public void run(){
    byte[] buffer = new byte[Const.MAX_HEADER_SIZE];
    while(true) {
        DatagramPacket receivePacket = new DatagramPacket(buffer, buffer.length);
        try {
            peer.mc.receive(receivePacket);
        } catch (Exception e) {
            System.err.println(Error.SEND_MULTICAST_MC);
            System.exit(0);
        }
        byte[] newBuffer = Arrays.copyOf(buffer,buffer.length);
        peer.executor.execute(new Runnable() {
            @Override
            public void run() {
                try {
                    interpretMessage(newBuffer);
                } catch (Exception e) {
                    System.err.println(Error.SEND_MULTICAST_MC);
                    System.exit(0);
                }
            }
        });
    }
}
```

Receiving a message and executing a thread to handle it

Backup subprotocol concurrency

In the backup protocol, the file to backup is first divided into chunks and then, for every chunk, a thread is executed to backup a given chunk. So that, backed files information does not collide, we used a *ConcurrentHashMap* to store information from every file a peer has started the backup. For every file, we store its name, fileID, intended replication degree, and a *ConcurrentHashMap* with chunk information. For every chunk, we store *ConcurrentSkipListSet* with the id of every peer that stored it.

```
ConcurrentHashMap<String, BackupFile> backedupFiles;
```

Backed up files *ConcurrentHashMap*

```
String name;
String fileID;
int replicationDegree;
ConcurrentHashMap<Integer, ConcurrentSkipListSet<Integer>> chunks;
```

BackupFile information

The thread created for each chunk sends the backup message, waits some time (according to how many times it has tried to send a message) and then checks if the number of stored peers for that chunk is at least the intended replication degree, if not tries again (up to 5 tries). The mc channel listener thread is responsible for updating the list of servers that have stored a given chunk every time a stored message is received.

From the side of the non-initiator peers, a backup message is received for every chunk, and if the peer decides to store that chunk (if he has enough memory left) the chunk information is stored on a *ConcurrentHashMap*. For every chunk the peer has its id, size, expected replication degree and the number of peers that also stored it. It also stores the chunk data in non-volatile memory.

```
ConcurrentHashMap<String, Chunk> storedChunks;
```

Stored chunks *ConcurrentHashMap*

```
String id;
int size;
int expectedReplicationDegree;
AtomicInteger storedPeers;
```

Chunk information

Restore subprotocol concurrency

When a peer receives a restore request, it creates an entry on a *ConcurrentHashMap* of restored files with the file id, file path, the number of chunks of that file and a *ConcurrentHashMap* where every chunk data will be stored. After that, it sends the restore chunk messages. When the MDR listener thread receives a chunk, it updates the restored file entry adding an entry to the chunks *ConcurrentHashMap*. When all chunks are present, the file is rebuilt in order.

```
ConcurrentHashMap<String, RestoredFile> restoredFiles;
```

Restored files *ConcurrentHashMap*

```
String path;
int chunksNo;
ConcurrentHashMap<Integer,byte[]> chunks;
```

RestoredFile information

Delete subprotocol concurrency

For a file delete request, the peer removes that file from the back up files *ConcurrentHashMap* and sends the delete message. Every peer that receives a delete file message removes all chunks it has from that file from the stored files *ConcurrentHashMap* and removes the chunk data from non-volatile memory.

Reclaim subprotocol concurrency

When a peer receives the reclaim space request, it updates its available memory. After that, it will remove chunks until the space it is using is below the available space. To decide which chunks to remove, it sorts them acording to how non-important they are (replication degree is higher than the expected replication degree) and the chunk size (bigger chunks are removed first). For every chunk it decides to remove, it sends the removed message to the MC channel.

Whenever a peer receives a removed message on its MC channel, it updates the amount of peer that have stored that chunk. If that number goes below the desired replication degree, it will start a backup for that chunk if no peer has started it yet.

Enhancements

BACKUPENH

To avoid depleting the backup space too rapidly, we decided to create a memory management method. Each peer knows how much memory it has available and how much memory it has being used. For this enhancement we call the subprotocol “BACKUPENH” with the same structure as the protocol based “BACKUP”.

With these values, we used the formula:

IF nTry = 5,

p(peer n saving) = 100%,

ELSE

p(peer n saving) = 0.1*nTry + 0.5*(free_space – chunk_size)/disk_space

nTry – Number of the try (1 through 4)

free_space – Space that peer n has free

chunk_size – Size of the incoming chunk to be saved

disk_space – Total space that peer n has

Note: For nTry = 5, p(peer n saving) = 100%

This formula makes it less likely for all the peers to store the chunk in the first try, which lowers the difference between actual replication degree and the perceived one, making memory less redundant but still allowing redundancy.

Moreover, a peer with a higher percentage of memory empty is more likely to store the chunk which makes the memory distribution along the network as even as possible.

RESTOREENH

In order to prevent overusing the multicast channels in the cases of large files, we developed a parallel way of transmitting the chunks through TCP.

A initiator peer that received a request from a TestApp with the subprotocol ‘RESTOREENH’ will send a message to the multicast control channel of a GETCHUNK like ‘RESTORE’ would do, but with version ‘1.1’ instead of ‘1.0’ and with a new line specifying the address and port where it will be listening for replies.

GETCHUNK <version> <senderId> <fileId> <chunkNo> <CRLF>

<ip_address> <port> <CRLF><CRLF>

A non-initiator peer, after receiving such message, will create a Socket to the ip address and port given. It will then write the message that would otherwise (in the version 1.0) send to the multicast data recovery channel.

The initiator-peer, after sending the GETCHUNK’s, will then create a connection with the properties sent in the GETCHUNK messages. Once it starts receiving connection requests, it executes a thread ‘ReatTCPAnswerThread’ for each request through the ThreadPoolExecutor.

```
if(enhanced){
    int counter = 0;
    int num = backupFile.chunks.size();
    while(counter < num) {
        try {
            connectionSocket = restoreSocket.accept();
        } catch (SocketTimeoutException e)  {
            if(restoredFiles.get(fileId) == null)
                return "RESTORED";

            else return Error.TCP_ACCEPT_CONNECTION;
        } catch (IOException e) {
            return Error.TCP_ACCEPT_CONNECTION;
        }

        executor.execute(new ReadTCPAnswerThread(this,connectionSocket));
        counter++;
    }

}
```

The thread will then keep reading through that socket and compile the chunk. After the whole chunk is compiled, it will then save it in a ConcurrentHashMap in the class RestoredFile which is kept in a ConcurrentHashMap called restoreFiles in Peer.java.

Once the RestoredFile notices it has all the chunks necessary, it restores the file in the proper directory and closes the connection.

DELETEENH

We developed DELETEENH in to prevent the storage of memory that is no longer desired by being offline in the moment that DELETE is called.

To fix this problem, we created two new messages that are only called in the enhanced version.

Greeter message:

HELLO <version> <peer_id> <CRLF><CRLF>

Deleted message:

DELETED <version> <peer_id> <fileId> <CRLF><CRLF>

The greeter message is always sent as soon as a peer connects to the network, this is made to let the rest of the peers know that the peer with peer_id has connected.

The deleted message is meant to be an answer to the message DELETE, saying that the peer with peer_id has successfully deleted the fileId.

Once a peer-initiator calls a BACKUP, it keeps track of which Peers saved chunks of the file. After it then calls DELETEENH, the peers with parts of the file saved, will respond with a DELETED after deleting those chunks. The peer-initiator will then update the data structure where it keeps the id's of the peers that had saved chunks of the file.

As soon as a peer that was offline during the DELETE message comes online, it will send the HELLO message. Every other peer, after receiving the HELLO message, will check if that Peer has any chunks that have already been commanded to be deleted. Therefore, if a peer with chunks from a prior DELETE file sends a HELLO message, the initiator-peer that commanded the DELETE message, will resend the DELETE message to make sure the newly online peer doesn't store unwanted information.