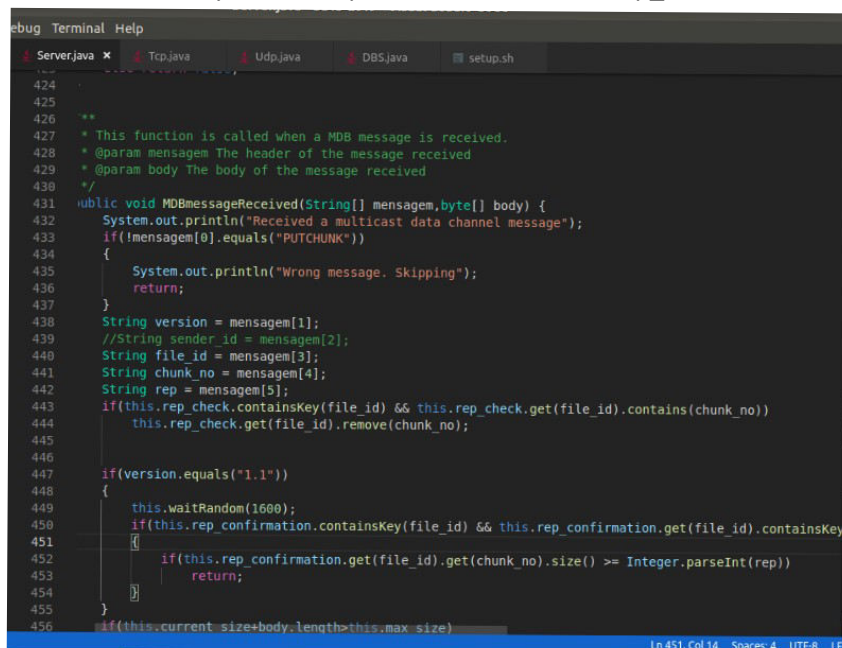


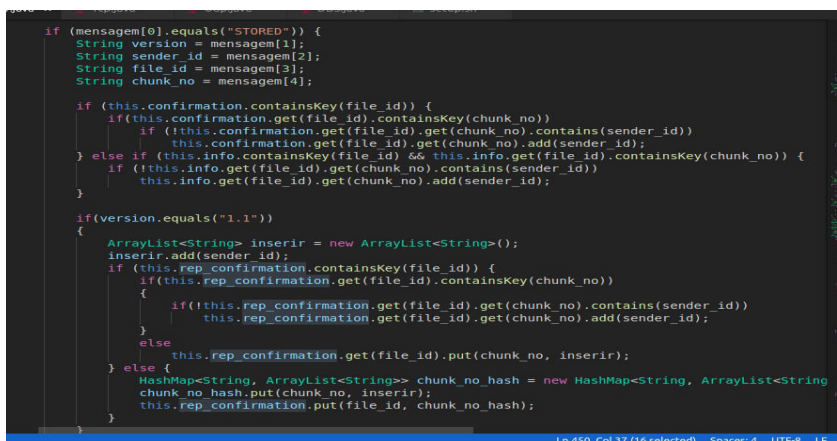
1^o Enhancement

The default scheme for the chunk backup subprotocol depletes all the backup space rapidly, since when we send a putchunk message in the MDB channel all the non-initiator peers will receive that message and save a copy of a chunk on it's file system, even though in a replication degree higher than the desired.

The ideal enhancement to this would be to try to guarantee that a chunk would only be stored x ($x == \text{rep degree}$) times, not higher and certainly not lower. To do this we created a data structure called 'rep_confirmation' that saves the server's ids that represent the servers that stored a specific chunk of a specific file and then on the function 'MDBmessageReceived', which is responsible to handle the putchunk messages received on the MDB multicast channel, we implemented a random wait of 0-1600 ms before saving the chunk and sending the message stored in multicast with an 'if' that would check if that putchunk is really needed, that is if the rep_confirmation data structure already has $x(\text{rep_degree})$ or higher servers that stored the chunk of this putchunk message. The importance of the random wait is to guarantee that not all servers try to save this chunk at the same time, or else this wouldn't work because they would all pass the 'if' test since rep_confirmation would be empty.



```
424
425
426 /**
427  * This function is called when a MDB message is received.
428  * @param messagem The header of the message received
429  * @param body The body of the message received
430  */
431 public void MDBmessageReceived(String[] messagem, byte[] body) {
432     System.out.println("Received a multicast data channel message");
433     if(!messagem[0].equals("PUTCHUNK"))
434     {
435         System.out.println("Wrong message. Skipping");
436         return;
437     }
438     String version = messagem[1];
439     //String sender_id = messagem[2];
440     String file_id = messagem[3];
441     String chunk_no = messagem[4];
442     String rep = messagem[5];
443     if(this.rep_check.containsKey(file_id) && this.rep_check.get(file_id).contains(chunk_no))
444         this.rep_check.get(file_id).remove(chunk_no);
445
446     if(version.equals("1.1"))
447     {
448         this.waitRandom(1600);
449         if(this.rep_confirmation.containsKey(file_id) && this.rep_confirmation.get(file_id).containsKey
450         {
451             if(this.rep_confirmation.get(file_id).get(chunk_no).size() >= Integer.parseInt(rep))
452                 return;
453         }
454     }
455     if(this.current_size+body.length>this.max_size)
```



```
if (messagem[0].equals("STORED")) {
    String version = messagem[1];
    String sender_id = messagem[2];
    String file_id = messagem[3];
    String chunk_no = messagem[4];

    if (this.confirmation.containsKey(file_id)) {
        if (this.confirmation.get(file_id).containsKey(chunk_no))
            if (!this.confirmation.get(file_id).get(chunk_no).contains(sender_id))
                this.confirmation.get(file_id).get(chunk_no).add(sender_id);
        } else if (this.info.containsKey(file_id) && this.info.get(file_id).containsKey(chunk_no)) {
            if (!this.info.get(file_id).get(chunk_no).contains(sender_id))
                this.info.get(file_id).get(chunk_no).add(sender_id);
        }

        if (version.equals("1.1"))
        {
            ArrayList<String> inserir = new ArrayList<String>();
            inserir.add(sender_id);
            if (this.rep_confirmation.containsKey(file_id)) {
                if (this.rep_confirmation.get(file_id).containsKey(chunk_no))
                {
                    if (!this.rep_confirmation.get(file_id).get(chunk_no).contains(sender_id))
                        this.rep_confirmation.get(file_id).get(chunk_no).add(sender_id);
                    } else
                        this.rep_confirmation.get(file_id).put(chunk_no, inserir);
                } else {
                    HashMap<String, ArrayList<String>> chunk_no_hash = new HashMap<String, ArrayList<String>
                    chunk_no_hash.put(chunk_no, inserir);
                    this.rep_confirmation.put(file_id, chunk_no_hash);
                }
            }
        }
    }
```

2º Enhancement

3º Enhancement

To make sure that all the peers delete the file they have backed up, we need to re send the delete message again until all the peers have deleted every chunk file.

To achieve this, we first need to save all the peers that have a certain chunk. So, when a putchunk message is sent the peers will respond with a stored message. All peers will save the stored message from that peers in the info data structure.

When they receive the delete message, they remove them self from the info structure (and delete the correspondent file) and send a removed message with the chunk number of “-1”. With this special message we can save a lot of messages from being send and the other peers know to remove the sender_id from the info structure.

There will be a thread (Delete.java) running every 1 minutes to check the info structure if there are files that weren't deleted. If it finds a peer that hasn't yet deleted the file, it will send the delete message. This will propagate through peers until the file is deleted. This event will happen every 1 minute.

If a peer receives a delete message of a file that has already deleted, it will still send the removed message so that peer know that he doesn't own the file.