SDIS 20/21 PROJ1 REPORT T5G04

All enhancements were made in the version 1.1 of the program and they are fully interoperable with the version 1.0 of the program.

1.1 Chunk backup subprotocol

This enhancement was implemented by stopping the listener of the data backup multicast channel blocking incoming traffic that would not have any effect on the peer's backup folder. When performing an operation that changes the available or maximum size of the peer, if there is any available space, the data backup multicast channel is reactivated.

Resuming a thread on reclaim or delete (lines 206 of process message and 425 of peer):

```
if (this.state.currentSize < this.state.maxSize && this.state.maxSize != -1
&& this.protocolVersion.equals("1.1")) {
    if (!this.dataListener.connect)
        this.dataListener.startThread();
}
```

Stopping the thread on putchunk:

```
try {
    if (this.peer.state.maxSize == -1 || this.peer.state.currentSize +
(msg.body.length / 1000) < this.peer.state.maxSize) {
        (…)
    }
    else if(this.peer.state.currentSize >= this.peer.state.maxSize &&
this.peer.state.maxSize != -1 && this.peer.protocolVersion.equals("1.1")){
        //if no more space is available stops listening to the data channel
        this.peer.dataListener.connect = false;
    }
```

1.2 Chunk restore protocol

When the initiator peer sends a GETCHUNK message, if both the initiator and the peer that is answering the message are on version 1.1, the CHUNK response, instead of having the chunk on it's body has a socket port and the

address is obtained from the DatagramPacket class. The peer that is responding immediately connects to the socket with a 2 seconds timeout in case the initiator does not connect. After the initiator peer connects to the socket, the other peer sends the body through that socket later closing it. If any of the peers is not in the version 1.1, the base version CHUNK message is sent.

Responding peer sending the port number (peer line 553):

```
//if the ServerSocket constructor has a '0', a free port is used
ServerSocket serverSocket = new ServerSocket(0);
//gets the port selected and sends it to the initiator peer
int boundPort = serverSocket.getLocalPort();
this.sendPacket("CHUNK", msg.fileID, msg.chunkNO, null,
Integer.toString(boundPort).getBytes());

//creates a timeout and connects to the socket. If the timeout is reached the
process is stopped
serverSocket.setSoTimeout(2000);
Socket clientSocket;
try {
    clientSocket = serverSocket.accept();//tries to connect
    BufferedOutputStream bof = new
BufferedOutputStream(clientSocket.getOutputStream());
    bof.write(body);//writes the body to the socket

    clientSocket.close();
    serverSocket.close();
    bof.close();
} catch (Exception e) {
    System.err.println("Noone Connected");
}
```

Initiator peer connecting to the port (MessageInterpreter line 106):

```

//gets the socket to communicate with the peer
socket = new Socket(msg.address, Integer.parseInt(new String(msg.body)));

BufferedInputStream in = new BufferedInputStream(socket.getInputStream());
byte[] buf = new byte[64000];
byte[] aux = new byte[64000];
int bytesRead, totalBytesRead = 0;
//since read can't read 64000 bytes, this stacks them in an auxiliary array
to copy to the body
//in order to keep the total bytes read intact.
while((bytesRead = in.read(buf))!=-1)
{
    arraycopy(buf, 0, aux, totalBytesRead, bytesRead);
    totalBytesRead+=bytesRead;
}

//copies the auxiliary array to the body array
body = new byte[totalBytesRead];
arraycopy(aux,0,body,0,totalBytesRead);
```

1.3 File deletion subprotocol

In this enhancement we created a new message DELETESUCESS that represents that the deletion was completed. This way the initiator peer has a new map that has the file id as the key and a set of the peers that should delete the chunks of the file as the value. After receiving the DELETESUCESS message, their entry from the set is removed. If there are any peers left on the set, when the initiator peer receives a message from a peer that has not deleted the file a new DELETE message is sent, expecting a DELETESUCESS message. In case the peer is in version 1.0, since there will be no response, the initiator peer sends another DELETE message to be sure the chunk is indeed deleted and removes the peer from the set right after, in order to not flood the channed with DELETE messages.

Sending new message (MessageInterpreter line 199):

```
boolean ret = this.peer.deletechunks(msg.fileID); //deletes all chunks,
return value is if successful or not
if(ret && this.peer.protocolVersion.equals("1.1") &&
msg.version.equals("1.1")){
 //additional response to the delete protocol
 this.peer.sendPacket("DELETESUCESS", msg.fileID, null, null, "".getBytes());
}
```

Processing deletesucess message (MessageInterpreter line 85):

```
if(this.peer.state.deletedFilesFromPeers.get(msg.fileID) != null) {
    this.peer.state.deletedFilesFromPeers.get(msg.fileID).remove(msg.peerID);
    if (this.peer.state.deletedFilesFromPeers.get(msg.fileID).isEmpty()) {
        this.peer.state.deletedFilesFromPeers.remove(msg.fileID);
    }
}
```

Concurrency Implementation

We decided to divide our project into three main classes, the peer, the message interpreter and the listerner.

Each peer has three listeners, one for each channel running in a thread. These listeners are responsible of listening to their respective channel and creating a message to be interpreted.

```
this.controlListener.startThread();
this.dataListener.startThread();
this.recoveryListener.startThread();
```

The message interpreter knows what to do with the received message making sure to call the correct function to complete the task. These functions create a new thread from a global scheduled thread pool so that concurrent operations are possible.

```
this.threadPool = Executors.newScheduledThreadPool(15);
```

In order to avoid busy waiting a scheduled thread pool was used instead of sleeps in these functions making the CPU resources available while waiting to complete the task as we can schedule a thread to be ran after a specific amount of time without wasting any resources in the mean time.

```
this.peer.threadPool.execute(() -> {
        //Function depending on the message
}, delay if needed)
```

During the start of the program, the peer also uses this scheduled thread pool when resuming operations that were canceled midway. These operations use similar functions to the original ones with some differences such as using the file id directly instead of the file path. (Peer line 124)

One thread from this pool is used in order to save the state of the peer in a thread that is executed every second in order to try to maintain the state as up to date as possible with minimal I/O operation as we thought that updating the state file every time the state changed is too CPU intensive as a file can have multiple chunks and multiple operations can be done at the same time.

```
peer.threadPool.scheduleAtFixedRate(() -> {
    try {
        peer.updateState();
    } catch (IOException e) {
        e.printStackTrace();
    }
}, 0, 1, TimeUnit.SECONDS);
```

We used a secondary scheduled thread pool when dealing with the backup protocol as every chunk of the file is dealt in a thread and this way the backup protocol would take up a majority of the threads available.

```
this.backupProtocolThreadPool = Executors.newScheduledThreadPool(5);
```

This new pool is only used in this protocol and nowhere else and as such we thought that a sleep wouldn't make much of a difference and so we used this in order to wait a second for the STORED responses.

Since the state of a peer is where everything important is stored, in the function responsible for the program's coordenation we declared it as "synchronized" so that only one access to it is allowed at the same time, mitigating the problems that two accesses at the same time could create.

(MessageInterpreter line 26)

```
synchronized (this.peer.state) {
        (interprets message)
}
```

Lastly, the peer implements the RMI interface and as such, multiple protocols and be executed at the same time without interfering with each other.

To attest our application fulfills the requirements for concurrency we tested various protocols at the same time.