Backup enhancement

The backup enhancement relies on the local database that each Peer keeps. Since each Peer attempts to track what is available on the system, whenever a PUTCHUNK is received they can check if enough STORED messages have already been received for the specific Peer and as such, stop themselves from storing the file and sending a STORED response message. The main function related to this enhancement is pictured below.

```java
/**
 * Handles enhanced BACKUP protocol by checking if desired replication degree has
 * already been met by other Peers.
 *
 * @param peer the singleton Peer instance
 * @param state the Protocol State object relevant to this operation
 * @return whether storing of chunk data should be aborted along with response message
 */
private boolean handleEnhancedBackup(Peer peer, ProtocolState state) {

    String hashKey = state.getFields()[Peer.hashI];
    int chunkHashkey = Integer.parseInt(state.getFields()[Peer.chunkNoI]);

    // Check that file hash exists
    if(!peer.getDatabase().getChunks().containsKey(hashKey)) {
        SystemManager.getInstance().logPrint("no data about " + hashKey, SystemManager.LogLevel.DATABASE);
        return false;
    }

    ConcurrentHashMap<Integer, ChunkInfo> chunksInfo = peer.getDatabase().getChunks().get(hashKey);

    // Check that chunk exists
    if(!chunksInfo.containsKey(chunkHashkey)) {
        SystemManager.getInstance().logPrint("no data about " + hashKey + "." + chunkHashkey, SystemManager.LogLevel.DATABASE);
        return false;
    }

    ChunkInfo chunk = chunksInfo.get(chunkHashkey);
    int desiredRepDeg = Integer.parseInt(state.getFields()[Peer.repDegI]);
    int perceivedRepDeg = chunk.getPerceivedRepDeg().size();

    // Update desired repDeg of this chunk
    chunk.setDesiredRepDeg(desiredRepDeg);

    SystemManager.getInstance().logPrint("perceived " + perceivedRepDeg + " chunk copies out of " + desiredRepDeg + " desired", SystemManager.LogLevel.DEBUG);
    if(perceivedRepDeg >= desiredRepDeg) return true;
    else return false;
}
```

Figure 1 - replication degree already met check

The database is update whenever a STORED, PUTCHUNK or DELETE message is received and as such is fairly up to date as long as the Peer is online.

Concurrency Design

The concurrency of the system mostly follows the suggestions on the course's webpage up to point 6.

The Peer maintains a ScheduledExecutorService for running protocols and timeout threads, as pictured.

```
private ScheduledExecutorService executor = Executors.newScheduledThreadPool(executorThreadsMax);
```

Figure 2 - the Peer's scheduled executor service field

Whenever the test client invokes a system protocol, an executor thread is created to handle the request.

```
executor.execute(new DeleteProtocol(filepath));
```

Figure 3 - test client requests threading implementation

If the request happens to be a RESTORE protocol then the invocation uses a scheduled timeout, which allows the RESTORE to run for a set amount of time before being considered that the file cannot be restored. The timeout is proportional to the total amount of chunks to be restored.

```
Future<?> handler = executor.submit(new RestoreProtocol(filepath));

executor.schedule(() -> {
    handler.cancel(true);
}, timeoutMS, TimeUnit.MILLISECONDS);
```

Figure 4 - RESTORE protocol scheduled thread

The thread running a RESTORE protocol instance gets interrupted by the "handler.cancel(true)" call and stops whenever it next polls the interrupted flag.

```
ConcurrentHashMap<Long, byte[]> chunks;
do {
    if(Thread.interrupted()) return false;
    chunks = state.getRestoredChunks();
} while(chunks.size() < state.getCurrentChunkNo());
```

Figure 5 - checking interrupted flag

As said before, the timeout threads also use this executor service to run the random delay before sending response messages. This is done by scheduling a thread to run after the delay as passed.

```
int waitTimeMS = ThreadLocalRandom.current().nextInt(Peer.minResponseWaitMS, Peer.maxResponseWaitMS + 1);
SystemManager.getInstance().logPrint("waiting " + waitTimeMS + "ms", SystemManager.LogLevel.DEBUG);

peer.getExecutor().schedule(new TimeoutHandler(state, ProtocolState.ProtocolType.BACKUP, this.channelName), waitTimeMS, TimeUnit.MILLISECONDS);
```

Figure 6 - timeout handler implementation

As for the message reception channels, each maintains an ExecutorService to run the message processing. This allows each channel to keep listening for new messages.

```
private ExecutorService executor = Executors.newFixedThreadPool(executorThreadsMax);
```

Figure 7 - the Service Channel's executor service field

As each message arrives, the listening thread submits a new handler to the executor service to run the processing.

```
    packet = this.listen();
} catch(IOException e) {
    SystemManager.getInstance().logPrint("I/O Exception on backup protocol!", SystemManager.LogLevel.NORMAL);
    e.printStackTrace();
    return;
}

// Submit to thread for processing
executor.submit(new SystemHandler(packet, this.channelName));
```

Figure 8 - submitting received message for processing

What additional threading could be done?

There are at least three ways in which the threading could be improved:

1. Usage of java.nio as suggested to keep the I/O non-blocking.
2. Usage of concurrent queues to keep the messages received and a separate thread would submit these to the channel executor service.
3. For BACKUP protocol different threads could handle a subset of the total chunks to backup. Same could be applied to RESTORE protocol.