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 1 - 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 2 - 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 3 - 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 4 - 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 5 - 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 6 - 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 7 - 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.