SDIS 2017/2018 - 2nd Semester
Project 1 -- Hints for Concurrency in Project 1

---

1. Introduction

This document suggests a path towards a highly concurrent and scalable implementation of Project 1. It uses as an example the implementation of the PUTCHUNK protocol.

The idea, as usual, is to develop gradually the protocol and through successive iterations improve the concurrency and scalability of your implementation.

Obviously, this document does not cover all the possible designs and you can certainly follow a different path. For example, unless you do not feel very confident with threads, you may want to skip the first design, and jump directly to Section 3.

2. Single Threaded Implementation

Assumption: this implementation allows the execution of only one protocol instance at a time.

By the end of this version, you will be able to handle the replication of files with a single chunk, or even with multiple chunks, as long as you start the replication of one chunk only when you are done with the replication of the previous chunk.

The idea is to use one single thread for each of:

- the initiator peer
- the non-initiator peers

Simplification: since this implementation is supposed to be a preliminary version, there is no need to measure the interval between retransmissions accurately. Instead of measuring this interval between retransmissions, measure the length of silence in the medium, i.e. either since the transmission of the most recent PUTCHUNK or since the reception of the most recent STORED.

This simplification allows the use of `DatagramSocket.setSoTimeout()` and `Thread.sleep()` to measure the passage of time, and use a single thread both for the initiator peer and for the other peers.

3- One Thread per Multicast Channel, One Protocol Instance at a Time

For this design, we still assume that there are no concurrent executions of protocol instances, i.e. at any time there is at most one protocol instance being executed.

The idea of this design is to use one thread per multicast channel for receiving the messages sent via the respective channel: the *control (channel) receiver thread* and the *(data) backup (channel) receiver thread*. (For the backup protocol, at least, you do not need a thread for handling the multicast data restore channel.)

Furthermore, each of these threads handles the messages received via the respective channel serially, i.e. they complete the processing of one message before starting the processing of the following message. Thus, each of these threads should execute an endless loop and in each iteration it:

1. receives a message on the respective multicast channel
2. processes the received message

The sending of PUTCHUNK messages by the initiator peer should be performed by a separate thread, the *multicaster* thread. This way, the multicaster thread may execute `Thread.sleep()` between the multicast of consecutive PUTCHUNK messages. This thread may be created in response to a request by the `TestApp` to execute an instance of the PUTCHUNK protocol, and may exit when that instance of the protocol terminates.

If you have implemented the interface between the `TestApp` and the `Peer` using RMI, the multicaster thread may be the thread created by the RMI run-time to handle the backup operation invoked by the `TestApp`.

Otherwise, i.e. if you are using either UDP or TCP for this interface, you will need an additional thread for receiving the requests from the `TestApp`. Like the receiver threads, this thread may also process the TestApp client requests serially: i.e., it may complete one request, before starting the following request by the `TestApp`. (Actually, if it supported the concurrent execution of `TestApp` requests, we would violate the assumption that there is at most one protocol instance executing at any time.)

Note that, in the initiator peer, the multicaster thread and the control receiver thread may have to share information, namely the number of received STORED messages. Thus, you must ensure that there are no race-conditions.

Note in this version, non-initiator peers need not process STORED messages. However, doing that is not much more complicated, because you must implement it for the initiator peer.

4- One Thread per Multicast Channel, Many Protocol Instances at a Time

I.e. in this version, we lift the assumption that there is at most one protocol instance in execution at any time.

We use the same thread design as in the previous version: there is a receiver thread per channel, and each of these threads completes the processing of a message, before starting the processing of the following message.

Although, there may be more than one multicaster thread, both on different peers and on the same peer, each of these threads participates only in one protocol instance execution.

In contrast, receiver threads may have to process messages of a protocol instance between the processing of messages of another protocol instance. Therefore, receiver threads must use one object per protocol instance for keeping the information relevant for processing messages belonging to that instance.

That object must be created upon the transmission, respectively reception, of the first message of its protocol instance, depending on whether or not the peer is the initiator of that instance. Furthermore, it must be kept in some data structure, and retrieved in order to process every subsequent message belonging to that protocol instance. Thus, the first action of a receiver thread after receiving a message should be to retrieve the object of the protocol instance to which the message belongs and then use the information in that object to process the message.

Hint: use a `java.util.concurrent.ConcurrentHashMap` object to keep the per protocol instance objects.

5- Processing of Different Messages Received on the Same Channel at the Same Time

In the previous version, messages received via a given multicast channel are handled serially: i.e. the processing of a message is completed before starting the processing of the following message.

To support the concurrent processing of different messages received on the same channel you need more than one thread per channel. A standard design, is to use worker threads: i.e. use a thread different from the receiver thread to process the received messages.

A simple solution is the receiver thread to create a new thread for the processing of each received message.

5.1- A more scalable solution

The problem with the above solution is that creating and terminating threads has some overhead, which is incurred once per message. A more scalable design is to use a thread pool, which allows to avoid creating a new thread to process each message.

So a new iteration of your design may be to use thread pools. In this case, you may want to use the class `java.util.concurrent.ThreadPoolExecutor`

6- No `Thread.sleep()`

The use of `Thread.sleep()` for timeouts can lead to a large number of co-existing threads, each of which requires some resources, therefore limiting the scalability of the design.

To avoid sleeping, you can use the class `java.util.concurrent.ScheduledThreadPoolExecutor`, which allows you to schedule a "timeout" handler, without using any thread before the timeout expires.

7- Eliminate all blocking

For ultimate scalability, you should remove whatever blocking calls are left, i.e. the calls for file-system accesss.

For that, you can use the class `java.nio.channels.AsynchronousFileChannel`.

9- Conclusion

This document focused on the concurrency aspects of Project 1, and implicitly assumed that the file being replicated had only one chunk. The suggestions presented here can be applied to the backup of files with more than one chunk, as well as to the implementation of the other sub-protocols.

As always, you should follow an incremental approach. However, in this case you have at least a two dimensional design space: in one dimension, you have the level of cconcurrency and scalability of your design, and on the other dimension the subprotocols. The number of chunks per file adds a third dimension.

Probably, the best approach is to follow a "depth-first" approach, in which you develop the concurrency of the backup protocol to the level you choose. After that, you can implement the other protocols with the same level of concurrency. However, you need to be careful and manage your time: if it takes you too long to take care of concurrency, you may not be able to implement all the protocols.

Further Reading

Surely, there are many resources about this subject on the Web. The following tutorials appear to me to strike a good balance between theory and practice and therefore may be useful in the design and implementation of SDIS's first project:

Java Concurrency / Multithreading Tutorial

A tutorial discussing concurrency issues in the context of Java. You may wish to take a look at the Concurrency Models chapter, which discusses several concurrency designs more abstractly than these notes.

`java.util.concurrent` Java Concurrency Utilitiels

A tutorial on the `java.util.concurrent` package, including `ThreadPoolExecutor` and `ScheduledExecutorService`, which is the interface provided by the class `ScheduledThreadPoolExecutor`

Java NIO Tutorial

A good tutorial on Java NIO, including the `java.nio.channels.AsynchronousFileChannel`