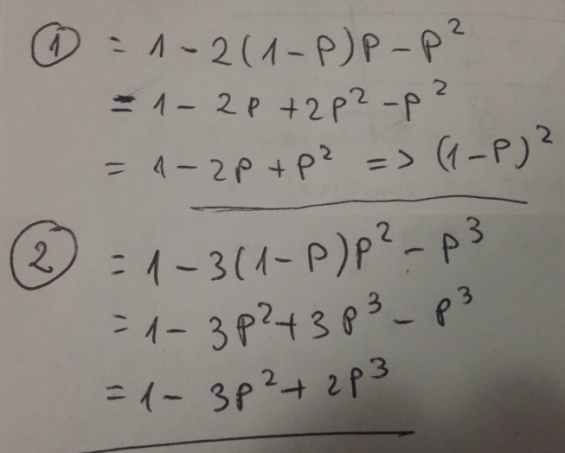
*Q1.1: What is the probability that the daisy chain network is connecting all the buildings?*

The probability that the daisy chain network is connecting to all the buildings is 100%, minus 2 times possibility of one chain failing minus the possibility that all chains fail (see picture 1).

*Q1.2: What is the probability that the fully connected network is connecting all the buildings?*

The probability that the fully connected network is connecting all the buildings is 100%, minus 3 times the possibility of one chain failing, minus the possibility that all chains fail (see picture 1).

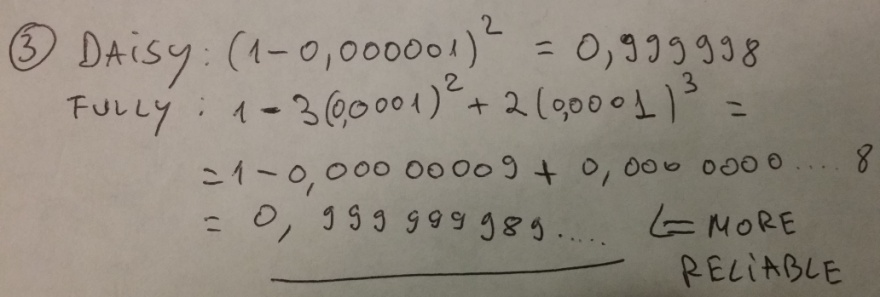


*Picture 1.*

Note: I am not sure though if I should also add “minus 2 times the possibility that one chain fails”.

*Q1.3: The town council has a limited budget, with which it can buy either a daisy chain network with two high reliability links (p = .000001), or a fully connected network with three low-reliability links (p = .0001). Which should they purchase?*

“Fully” – see picture 2.

**

Picture 2.

*Q2. A three-phase commit protocol has the following parts:*

*Phase 1: is the same as for two-phase commit.*

The 2PC is a blocking protocol, it means that it blocks each node until a commit or roll-back is received. If the coordinator fails after a participant sent an agreement, the participant will block and lock the objects until the coordinator is recovered and starts the operation again.

If the coordinator waits for another participant to return tell that everything is fine, but the participant fails, the coordinator will wait and other participants will be left blocking. In order to solve this issue a time-out can be inserted for the coordinator so that the coordinator sends an abort to the remaining participants allowing them not to be blocked.

The 3PC protocol on the other hand prevents the delay due to blocking by introducing an extra step or sending the commit. If the coordinator fails while or before sending the commit, the participant will assume an abort after the time-out is finished and will roll back and release all locks, after that it will continue with the remaining transactions. This is not possible in 2PC, because if a coordinator already sent a commit to some participants and then failed the system would not be in a consistent state as there will be a possibility that some nodes committed the transaction and others aborted it (if a time-out was implemented).

***Programming***

***Question 1***

**For *certainBookstoreReplicator*** we created two global variables, one HTTP client and one executor service. In the constructer we initialized the executor service with the maximum number of replicator threads. In the constructor we also initialized the HTTP client with the number of threads and max connections per address equal to maximum replicator threads. This was done to ensure that we can handle all the executor services.

A timeout was also set in case we have an error, so the client will abort in order for other clients not to be blocked. The client was also started. This is the fail-stop model we used to handle failure of system components. The only problem with this strategy is that a slow-running system will appear to be stopped but it is easier to implement.

In *replicate* method we created “List<Future<ReplicationResult>>”, and for each slave server in the input set the CertainBookStoreReplicationTask was started in order to pass the slaveServer, request and client. The future tasks were also generated based on the replication task, and they were executed. The future tasks are inserted into the full list that is returned in the end of the method. By doing this the latency hiding is achieved, because the list of future tasks is sent together with request for replication so the slave server starts solving the requests together with the replication process.

**For CertainBookStoreReplicationTask** we changed the constructor to take as input the *slaveServer,*for the *request* and the *HttpClient* as these are needed to send back the answer for the request.

Inside the method *Call* we prepare the data needed to send the message, by inserting the request into XML and turn into a byte array which can be sent. We also create the URL at which the message will pe sent. In next stage we created the Content to enable the communication and at the end the message is sent, and an answer is waited. The *ReplicationResult* is return with the field *replicationSuccessful* set to be *false*.

**For the replication process on the slave server** we have added all the methods to handle replication requests, update the bookstore contents and modify the snapshot identifier: *addBooks*, *addCopies¸ updateEditorPicks, buybooks*, *removeAllBooks* and *removeBoooks*. Inside of each method we called the correspondent function from *CertainBookStore* incremented the snapshot ID and wait for result that is returned.

In the **SlaveBookstoreHTTPMessageHandler** we created handlers for each type of request to server which we implemented almost the same like the other handlers that were present in the *switch*.

**For handling the replication on the proxy servers** we have chosen to make a random selection of building block. For this we generated random numbers to choose which proxy or if the master will be chosen for the replication.

***Question 2***

The system we implemented can handle properly all concurrent requests that read data from database which is replicated on slaves and the master sends requests to the slave to modify the data properly. The bottleneck is at the master server which has to handle all requests for updating the store.

***Question 3***

The client reads cannot go back in time because the proxy is always waiting for a result whose *snapshotID* is larger than the previous one.

***Question 4***

In that case the master will not be able to replicate the results on separate servers in case of a write request which will lead to inconsistences concerning the snapshots.