**1. Statement of the Problem**

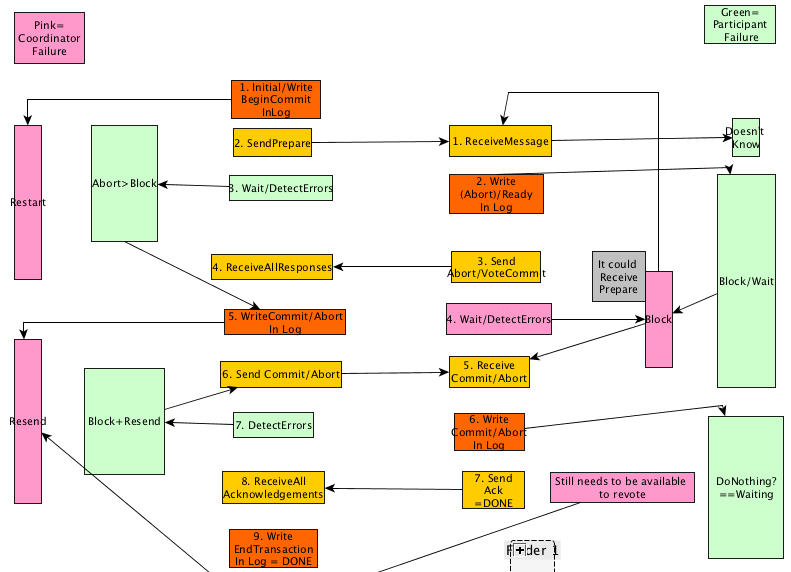
The two phase commit (2PC) protocol is an atomic commitment protocol which is a specialized type of consensus protocol. The goal of the protocol is to atomically commit / store a new value at all of the nodes in a distributed system that store that value so that all nodes are synced to the new value, even if there is node failure or network failure during the commitment process. I implemented 2PC in Java using the RMI library. I simulated site failures at various points during the commitment process, but did not simulate network failures.

**2. Methods and Procedures**

Architecture: The code is written in Java using the RMI library for communicating between sites. There are two remote interfaces (coordinator and participant). The interfaces implement methods like "Participant.receivePrepare" and "Coordinator.ReceiveVote('commit')".

Algorithm: To understand the algorithm, I started by reading sections 12.4 and 12.5 of the textbook "Principles of Distrubted Database Systems" by Ozsu and Valduriez.

The Ozsu textbook came with a helpful diagram of the 2PC algorithm, but I augmented my understanding of the algorithm by creating two helpful diagrams.



Should the image above be too small to read, it can be found at the following url: <http://i.imgur.com/ENXoNnl.png>

Limitations and Capabilities:

* One limitation of my 2PC limitation is that site failures are only detected by catching errors thrown when invoking a method on an RMI stub (as opposed to catching errors with timeouts). As a result, the coordinator never "knows" that a participant has failed until the participant has recovered and re-connected with the coordinator. This is not an issue in 2PC, since the Coordinator usually blocks in response to participant failures anyway.
* However, before I decided to implement 2PC I was working on 3PC, and 3PC is a significantly more complex algorithm to implement, since it requires the use of timeouts, a leader election algorithm, and more state transitions. Furthermore, 3PC doesn't always lead to correct results in the case of network partitions.

**3. Data Collected**

The diagram linked in the above section shows 9 states that the coordinator can fail in in and 7 states that a participant can fail in. One can thus enumerate the discrete space of possible states that a single (or multiple) failure(s) might occur in. To simplify the analysis of my implementation, I only simulated failures in states 3 and 7 of the coordinator and state 4 of the participant.

Failures could be simulated in a variety of ways. One could simply add thread.sleep() calls into the code, and then kill a process from the command line to simulate a failure. Another option would be to hard-code a System.exit() call into the code. I created a better solution than both of these. The code is written so that in order to proceed with the algorithm, the user must give interactive input from the command line. The user can crash the process at anytime. In all cases, recovery is done by manually restarting the process from the command line.

A sample log is included in the github repository containing the code.

**4. Experiences**

I learned a lot about Java in the process of implementing 2PC.

I had to learn about implementing java threads in order to have the coordinator communicate with the participants. Since Java is synchronous, it blocks on all remote method invocations. Thus, if the coordinator invokes a remote method on a client, which in turn invokes a remote method on the Coordinator, a deadlock could occur. Java RMI does not guarantee that RMI calls will or will not be given a new thread, so one cannot guarantee that a deadlock will or will not occur.

In order to implement the threads in Java, I learned about nested anonymous classes (NACs) in java, since they are the most succinct way of creating new threads (in my opinion). I had to create several different nested anonymous thread classes in order to send the different messages ("prepare","commit","abort"). Were I using Java 8, I could have used it's functional programming features to pass a function to a single thread class. This would have led to much more succinct code in the "createThread" method.

I also learned about several interesting scoping issues that NACs have. To refer to the enclosing parent instance, you use "ParentClass.this.method();". To reference (from within the NAC) local variables from the enclosing parent class, you must declare them final in the parent class. This is because Java copies (into the NAC) these local variable values by value rather than by references.

For a while, I was stuck, since I wanted to implement voting by using a single counter (e.g. the Coordinator would send "votePrepare" and the client would send "voteCommit" and the coordinator would do "voteCommitTally++"). However, this is not robust to failures. The same client could fail, reconnect and vote several times. If remote reference objects were mutable, I could have set "participant.votedCommit=true", but they are not. Ultimately, I choose to use a "votedCommitArray" and to hard-code the index for each participant. This has several drawbacks, like being memory inefficient, and requiring fixed size arrays, but these seemed like minor drawbacks.

**5. Observations**

Trying to understand the algorithm was harder than it should have been. The Ozsu textbook wasn't sufficiently clear, so I also looked at the the presentation of 2PC in the textbook "Distributed Systems" by Tanenbaum (which was also insufficiently clear). I ultimately resorted to creating my own 2PC diagram so that I could understand the algorithm.

My diagram has several advantages over the presentations in the other textbooks. Most importantly, it visualizes not only the failure-less execution flow chart (as in Ozsu), but it also visualizes how the algorithm behaves when there are failures in either the Coordinator or the participant. Visualizing the execution of 2PC without failures is completely missing the interesting behavior of the algorithm.

Another advantage it has is that it is color-coded, so that it is easy to see when the coordinator or participant writes to it's stable storage log (and simultaneously, how the behavior of it's recovery protocol changes.). The color coding also makes it easier to see how the steps of the recovery protocol for participant correspond to the termination protocol of the coordinator (and visa versa).

There is also one particular detail that seems to be explained poorly in both textbooks. When the coordinator detects a timeout/failure in the Wait state, the Ozsu textbook explains: "In the WAIT state, the coordinator is waiting for the local decisions of the participants. The coordinator cannot unilaterally commit the transaction since the global commit rule has not been satisfied. However, it can decide to globally abort the transaction, in which case it writes an abort record in the log and sends a “global-abort” message to all the participants. " However, it could just as well block and wait for the participant to come back online. It doesn't explain why it makes one (seemingly arbitrary) choice over another.