Distributed Transactions: Solution Strategy

Jonathan Shaw

February 10, 2020

1 Introduction to Problem

Given a distributed system involving many processes executing across many nodes, suppose that one node initiates a transaction in which the other nodes are involved. In order for the transaction to take place, *all* nodes in the system must be able to commit. In the absence of shared memory, how do we decide whether the transaction, as a whole, is committed or aborted?

As it happens, there is a very nice algorithm called **Two-phase Commit** by which a specific coordinator node, or **transaction manager**, mediates between the initiating node and the other nodes and is charged with having the final say over whether the transaction is committed. Therefore, the purpose of this project is to provide a visual, interactive, and scalable simulation of transaction consensus via the Two-phase Commit algorithm.

2 Two-phase Commit

Before explaining the two-phase commit algorithm, we will quickly recap some important definitions.

- Our system consists of a collection of processes called **resource managers (RMs)**, each of which executes on a different host. One special node in the system is designated as the **transaction manager (TM)**.
- A **transaction** is an operation performed by multiple resource managers. In our simulation, it will be performed by *all* present RMs.

Each RM has four possible **states**, and an RM can be in only one state at a time. The possible states are *working*, *prepared*, *committed*, and *aborted*. The states are governed by the following set of rules:

- The "default" state for each RM is *working*. At the beginning of the simulation, all RMs should be in the *working* state.
- An RM in the *working* state may enter the *prepared* state or the *aborted* state.

- An RM in the *prepared* state may not change state until it is signaled to do so by the TM.

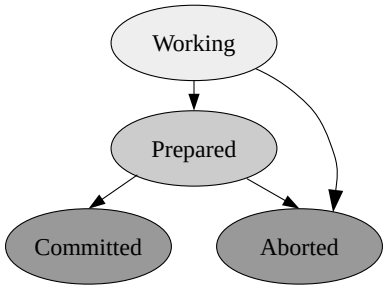

Figure 1: State Diagram for Resource Managers

We will now proceed to describe how the algorithm proceeds. Rather than writing pseudocode, we will explain the procedure in prose.

When an RM commits a transaction, it enters the *preparing* state and alerts the TM, via a message, that it has done so. Upon receipt of the message, the TM initiates the Two-phase Commit protocol.

Phase 1: Preparation

1. The TM sends a *prepare* message out to all RMs in the system.
2. Upon receipt of the *prepare* message, each node enters the *prepared* state if it is able. If it is not able to enter the *prepared* state, it enters the *aborted* state.
3. Upon changing state, each node notifies the TM (via a message) of its state change.

Phase 2: Commit/Abort

1. If the TM finds that *all* RMs have entered the *prepared* state, it notifies each RM to enter the *committed* state. The transaction is now complete; it has been committed.
2. If, on the other hand, *any* RM has entered the *aborted* state, the TM notifies *all* RMs to enter the *aborted* state. In this case, the transaction has been aborted.

3 Assumptions (updated)

In my problem characterization document and solution presentation, I laid out some assumptions that I would make about the system. Upon doing more research and listening to my classmates' presentations, I have updated the project assumptions:

- Previously, I made no assumptions about network topology other than connectedness. However, since message routing is not the problem I am discussing, I will now assume that all RMs are directly connected to the TM.
- Each "host" in the system will run exactly one RM, so that the system can be viewed as a graph of RMs. The only exception is the TM "host", which will run only the TM process.
- The channel will be assumed reliable; messages will not be dropped or misinterpreted.
- Since fault-tolerance is not the problem of my project, I will assume that all RMs remain connected and respond appropriately to all messages.

4 Implementation

4.1 Project Goals

I have set out the following criteria that I would like my project to meet.

1. The program should be able to run, with minimal dependencies, on all standard desktop/server operating systems.
2. RMs should run on separate threads and should communicate exclusively through message passing.
3. The program should be able to be run on a single machine or over a network; that is, it should be possible for RMs on one host to communicate with the TM on another machine.
4. My simulation program should be interactive and visual. Using the program should be self-explanatory.

4.2 Technologies

- **Platform:** To achieve the first goal, I will write my simulation program in Java. Since the Java runtime environment is present on most devices, compatibility should not be an issue whatsoever.
  - Java was the first programming language I ever learned, and is still the one I have used far more than any other. I am quite comfortable with this language—this is a major benefit to me, as I am currently taking six classes and so I have little time to dedicate to learning a new language.
  - While it does not have as many fancy libraries as something like Python, I am confident that the power and flexibility of Java will be more than sufficient for my needs.

- **Threading:** I intend to use the standard `java.util.concurrent` library to run the TM process, as well as each RM process, as a separate thread. All threads will be concurrently active.
  - Due to my joint major, I have not taken a course in concurrency and threading. Nevertheless, I have had *some* experience using threading in Java—for example, I have written a program whereby two matrices are multiplied and each entry of the product is computed in its own thread. I do not expect that figuring out the concurrency aspects of this project will be a major challenge: if I do run into problems, I will seek advice from a student in CPSC 222.
- **Communication:** The `java.net` package should provide all the message-passing capabilities I need. In particular, I plan to use the `java.net.ServerSocket` and `java.net.Socket` classes to establish a TCP connection between the TM and each RM.
  - Again due to my joint major, I have not had any experience whatsoever in network programming. For this reason, I expect communication between nodes to be one of the most difficult parts of the project to figure out. From my own readings, I am somewhat familiar with computer network concepts—this gives me the vocabulary, at least, to ask questions and pursue further individual study in this area. If I find I am really struggling in this area, I am in good hands: Dr. Hakak has told me that he is more than happy to give advice related to computer networks and communications.
- **Interface:** I will use the `java.util.swing` widget toolkit to build a graphical user interface.
  - Widget toolkits like Swing are conceptually easy to work with, but it can be tedious to actually design and implement a graphical user interface. I expect that creating an interface will take a significant quantity of my time in this project. I may be able to speed things up in this area by using a graphical tool rather than writing all the Swing code by hand.
  - Since the GUI will probably take some time to develop, I will start by first designing a simple CLI-based interface. By supplying an argument, the user should be able to select the interface to be used when the program starts (akin to Emacs).
  - If I run into problems with Swing, I will consult with the documentation.

4.3 Architecture and Design

The two classes of major importance for this project are the RM and the TM. These components will be assembled in what is approximately a client-server architecture: the TM acts as the server, whereas the RMs act as clients. I will provide a brief description of these classes:

4.3.1 Transaction Manager

Upon instantiation, the Transaction Manager should be provided with a port number upon which it will listen for incoming TCP handshakes. It should maintain a list of sockets through which it is connected to RMs. The TM should simply continually check its port for messages and broadcast messages to each of the connected RMs when necessary.

4.3.2 Resource Manager

Upon instantiaion, the RM class should be bound to an arbitrary port and should be given the TM socket. It should immediately connect to the TM, and then listen for messages and behave accordingly according to the algorithm in Section 2. Additionally, the RM class should have a method to initiate a transaction—the user should be able to invoke this message.

Resource managers should have a field that indicates the probability that, when asked to enter the *prepared* state, they instead enter the *aborted* state. The user should be able to manipulate this parameter (global across all RMs) to produce simulations as he/she sees fit.

4.4 User Interface

I have drawn up a rough sketch (Figure 2) of how I envision the GUI for my program. The following is a brief description of the various GUI components:

- The "New Simulation" button returns all existing RMs to their default state where no transaction has yet been initiated. This button will also wipe the log panel.
- The "RMs" spinner allows the user to instantiate or de-instantiate RMs, which will be connected to the TM and appear in the display panel. Below this spinner, the number of RMs in each state is shown.
- The "$p(\text{abort})$" box allows the user to enter the probability that a node, when asked to to enter the *prepared* state, instead enters the *aborted* state.
- The Log panel shows in real-time what is going on from the transaction manager's point of view.
- The display panel gives a visual representation of the system as a graph. The user is able to click an RM to initiate a transaction from that RM. The RMs and TM will visually change in color based on their state, and perhaps the connections between nodes will visually indicate when messages are being sent.

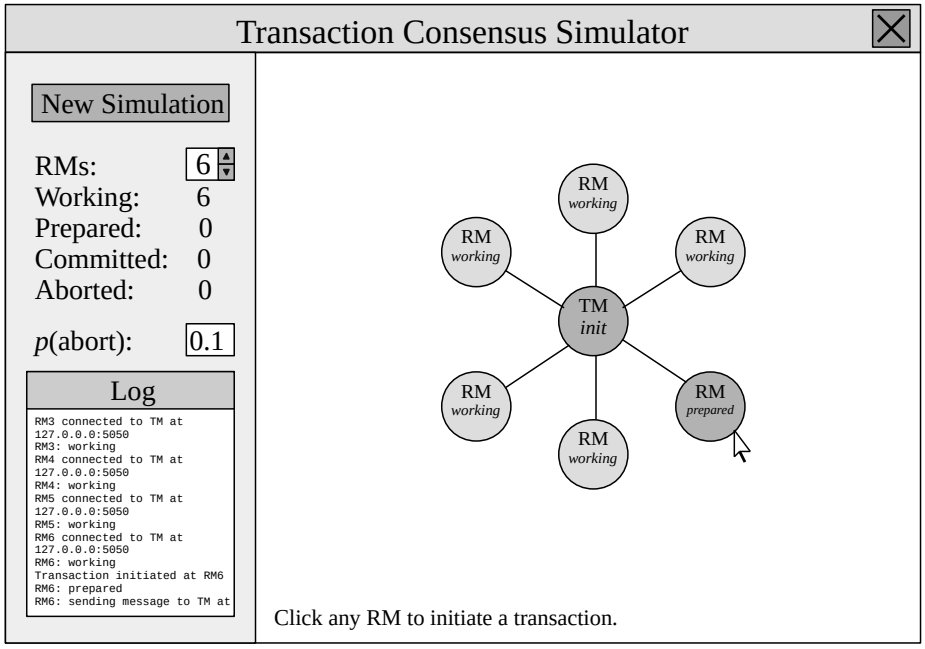

Figure 2: Rough mockup of GUI

5 Development Plan

5.1 Major Milestones

When thinking about the development process ahead, I imagine the following four events as the major markers of progress:

1. **Creating processes on two threads and getting them to affect each other by message-passing over sockets.** This is likely the biggest milestone in the project, since I am relatively new to concurrent programming and completely new to network programming. This milestone is critically vital: without reaching this milestone, I have nothing to show for my project.

2. **Creating RM and TM classes and implementing the algorithm.** Once the nitty-gritty details of communication and threading are resolved, the implementation of the Two-phase Commit algorithm should be relatively simple. Nevertheless, it marks a major milestone in the project, since it indicates that I have solved the problem assigned to me. This milestone is vital to the project, as it represents the absolute bare minimum program that could actually be considered valuable.

3. **Creating a GUI for the program.** While the GUI is less vitally important to the project than the first two milestones, it is still instrumental for the goals I laid

out. I expect the GUI portion of the project to actually take longer than the core implementation component, as it can be quite cumbersome to work with widget toolkits.

4. **Making the program work over a network:** If I have time, I would like to modify the program so that RMs can be instantiated on one machine and, over a network, communicate with the TM on a different machine. I don't expect that this should be too difficult to do with the port interface, but given my lack of experience I could be incorrect.

5.2 Timeline

Week 1: Feb. 9 – Feb. 15

As I have four midterms this week, I will not be working whatsoever on the project.

Week 2: Feb. 16 – Feb. 22

This week is reading week. I am lucky this semester, as my midterms and major assignments fell mostly before the reading break—as such, the reading week remains relatively open for me. During this week, I hope to hit both the first *and* the second milestone discussed in the previous subsection: this would give me a nice product to submit for my first intermediate report.

Week 3: Feb. 23 – Feb 29

I will consider this week to be a sort of "buffer"—if I don't meet my goal of hitting the first two milestones in Week 2, I will work on them in this week. Otherwise, I will begin planning the GUI implementation.

Week 4: Mar. 1 – Mar. 7

I hope to work heavily on the GUI during this week. My goal is to get it to a semi-functioning state.

Week 5: Mar. 8 – Mar. 14

I will take a break from the project during this week, as I have several midterms.

Week 6: Mar. 15 – Mar. 21

Work on the project will be be light during this week, as I continue to have midterms and projects in other courses. Nonetheless, I hope to hit the third milestone by the end of this week. I will submit the second intermediate status update document, along with code that, hopefully, has a functioning user interface.

Week 7: Mar. 22 – Mar. 28

If I do not hit the third milestone in Week 6, I certainly plan to meet it this week. Assuming the third milestone has been hit, this week should be reserved for refactoring, tweaking, and thinking about the fourth milestone.

Week 8: Mar. 29 – Apr. 4

During this week, I will tweak the project and continue to work towards the fourth milestone if it has not been hit. By the end of this week, I should be able to say "this is a finished product".

Week 9: Apr. 5 – Apr. 11

I will submit my finished product and begin my code review of Gaurav's project. The code review will be finished and submitted by the end of the week.

Week 10, 11, 12: Apr. 12 – Apr. 24

Amidst my studying for other exams, these weeks are the time for me to prepare my final project presentation. After presenting, I will have finished everything related to the project.

5.3 Potential Challenges

The following challenges could arise, adding significant difficulty to the project.

1. Since I have no experience in network programming, it is possible that the solution I have proposed is deficient in some way. If that were the case, I would have to learn new tools in order to complete the project.

2. When working with graphics, special care needs to be taken to avoid "spaghetti code". To avoid having an over-abundance of pointers, I may have to learn how to use event listeners and other related tools to stimulate GUI changes.

3. When trying to reach the fourth milestone, I may find that firewalls prevent my processes from communicating over some networks.

Those are challenges that I am aware of—likely, however, I will be faced with challenges I haven't even thought of. In my timeline, I have budgeted a reasonable amount of flexible time that I can use if some milestones take longer than expected to complete. In the worst case-scenario, I can abandon the fourth milestone and still turn in a very complete project.