Two phase commit:

Protocol:
The protocol between the server and the user node is explained below.

```
Server                                   Node

1. prepare -------------------------->

2.<------------------------------------ user decision

3. server decision ------------------>

4. <----------------------------------- Acknowledgement
```

On the start of a new commit, the server creates a commit object ,which contains all the information necessary for a commit to take place, such as the usernode list, the filenames, the collage bytes etc. Each commit is identified by a unique integer that is used to store the commit object in a hashmap that is used in the server in order to identify which message corresponds to which commit.

The commit object then sends a prepare message to each individual usernode involved in the commit with the byte array containing the collage, the list of files pertaining to that particular usernode, the commit id etc. All this information is encapsulated into a 'message body' object which is then serialized to be sent via a ProjectLib message.

The usernode on receiving the prepare message, invokes the askuser method with the relevant information and passes on the user's decision to the server. However, before asking the user, the usernode makes sure that any file that is currently being requested for a commit has not been used for any other commit operation by checking the file list it maintains.

The server gathers all the user decisions from the usernodes. If any node says no, it passes the serrver decision with abort commit message to all the usernodes for that particular commit. Each usernode sends an acknowledgement for this message.

Timeouts and Lost messages:
The timeout for each message at the user side is kept at 6.5 seconds. For each prepare/server decision message sent at the server, a new thread is started. That thread sleeps for 6.5 seconds and checks the hashmap of userdecisions/ acks to see if the reply had arrived for that message . If an entry in the hashmap is not found, the thread either updates the decision pertaining to that node to be a false one, or if it is a server decision message, it sends the server decision message again in intervals of 6.5 seconds till it finall receives an ack for that server decision message.

It is to note that all messages in the server are received by the main thread and each commit object pertaining to that commit id is updated only by that thread. Hence timers on the other threads can just check the pertaining data structures and take the relevent action.

Recovery from Failures:

Server side:
A log file is maintained for each commit. It is named as <commitId>.log. The first line in the log file contains the collagefilename and a comma seperated string of all the sources. Every other line is a key value pair with the key as the type of message and the value being the usernode that the message is sent/received from. The type of messages were explained while describing the protocol above. At startup, the server reads all the log files, staring from 1.log to etc.. till it encounters a missing file, at which point, the state restoration is complete. After reading the entire log file, if the server finds that the commit is incomplete without a receiving user decisions, it aborts the commit by sending a serverdecision message with an abort flag. If the server finds that it had received all the decisions from all the relevant usernodes, it makes the final decision and then sends its decision one more time to all the relevant usernodes one more time.

On every message sent/received the server writes to the log file in the corresponding commit file, based on commit id.

User Side:
All the files that were deleted as a part of a successful commit operation are logged in a file called <usernodename>.log . This log file is read from, at the startup of each usernode and the deleted file list is updated in order to ensure correctness of future commits. Writing to the log file is done every time a file is deleted as a part of every successful commit.