Code structure and interfaces)
The project was implemented using java rmi and is seperated into a Client, Coordinator, and ReplicaServer. ReplicaServer and Coordinator both take one argument for the name of the server on the command line. Client takes no arguments.

Client takes user input and can issue a stream of commands to the Coordinator. Coordinator does the bulk of the two-phase commit protocol by issuing vote requests and commands to the replica servers. ReplicaServers keep track of active transactions, submits queries to its own sqlite database, and prevents concurrent modification of the same keys. Both the Coordinator and ReplicaServer keep track of their own logs and a ReplicaServer may ask a Coordinator or another ReplicaServer the outcome of a transaction.

There are three ReplicaServers with each running on n0004, n0005, and n0008 respectively and each have their own log and database (<server name>.db or <server name>_log.txt on the NFS). There are two Coordinators with each running on n0000 and n0001 respectively and they both share a log on the NFS (n0000_log.txt). The n0001 Coordinator is just a backup to n0000 and running it is completely optional. If n0000 is down then n0001 will receive and issue commands from the client while logging to the shared log.

*All three replica servers must be up for the coordinator to issue commands.

There is a Coordinator interface and a ReplicaServer interface. The coordinator exposes get, put, and delete which are rmi methods invoked by the client and it exposes getDecision which is used by the replica server to get the outcome of a vote. I originally had the Coordinator inteface split into one for the client and one for the replica servers but I merged them for the sake of simplicity.

Coordinators will log its initialization, the start of a transaction, the outcome of a vote, and if it has finished attempting to send the outcome of a vote to all the replicas for each transaction. Replicas will log its initialization, its vote for a transaction, the outcome of a transaction, and if it has finished commiting a transaction to the database for each transaction. These logs are written before their respective actions are done except for the completion log which just says a transaction is done.

So coordinators and replica servers may have the following statuses for each transaction in their logs.

Coordinator: start, outcome, done.
ReplicaServer: vote_<commit or abort>, outcome, done committing.

Recovery is done for the Coordinators and Replica servers at initialization before they expose any rmi methods. Recovery is done by reading each line of the log and pushing it to a stack and then parsing through each line in the stack (this is a simple way for reading the log from end to start). The parsing of the lines uses the transaction statuses above to decide on what to do during recovery.

For coordinators done transactions found in the log are added to a hash set of transaction ids for completed transactions. If the recovery mechanism finds a line with a transaction id not in the hashset it will do one of two things. If the transaction has an outcome it will resend that outcome to all the replicas and log it as done or if it has only just started then it will log an abort, tell the replicas to abort the transaction, and then log the transaction as done.

For replica servers aborted transactions or transactions done commiting found in the log are added to a list of complete transactions. If the recovery mechanism finds a commit outcome for a transaction but sees that it never finished then it will redo the transaction, commit it, and log it as done. If a vote to commit is found and the transaction was never completed then the replica server will ask the coordinators and then its peers (if the coordinators are unavailable) for the result of the transaction. If the transaction is active it will redo it and then wait for an outcome, if it is aborted it will just log it, and if is commited it will log the outcome, redo the transaction and commit it, and then log it as done.

Basically every start/vote and outcome line for a transaction id will attempt to be matched with its respective done line in the log. If there is no match then some recovery action will take place.

The current transaction id is also calculated from the log (the highest transaction id so far plus one) for future transactions.

* This log reading implementation is not scalable for large logs.

A scheduled task that runs every 3 minutes on the replica servers will find out the outcomes of active transactions and try to take necessary actions.

RMI interface replicas expose to coordinator)

```
public String doGet(String key) throws RemoteException;
```
Coordinator chooses a random replica to do this. This just returns the value for the key otherwise it says it does not exist.

```
public boolean doPut(Transaction t) throws RemoteException;
```
Creates a lock for the associated key (if necessary) and tries to grab it, creates a transaction on the data base, attempts to insert the kv pair into the data base, and then adds the transaction and its connection to a list of active transactions. If any of these steps fail an error handler will be called which cleans up any resources and false will be returned.

* This method can be removed from the interface as it is never called remotely. This could also be consolidated with doDelete.

```
public boolean doDelete(Transaction t) throws RemoteException;
```
Does the same thing as doPut except attempt to delete the kv pair from the data base instead of inserting it.

* This method can be removed from the interface as it is never called remotely. This could also be consolidated with doPut.

```
public boolean canCommit(Transaction t) throws RemoteException;
```
Call doPut or doDelete (they should be consolidated and changed to static… made a todo on it in the code) and then log the result and return it to the coordinator.

```
public void doCommit(Transaction t) throws RemoteException;
```
Log the global commit, commit the transaction and clean up resources, and then log the transaction as done committing.

```
public void doAbort(Transaction t) throws RemoteException;
```
Log the global abort and then abort the transaction and clean up resources.

```
public TransactionStatus getDecision(Transaction t) throws RemoteException;
```

Scan the log for the transaction in the same way the recovery mechanism does it and return the status as committed, active, aborted, or nonexistent.

Failure detection and visibility to client)

Failures for most of the rmi methods are done by returning a string or a boolean value at the end of a rmi call or when an exception is thrown. If any modifying transaction fails due to database integrity constraints then client just sees that the command failed. But if a failure occurs due to one of the systems failing in the middle of a modifying transaction then the client sees a message that the command may or may not have failed due to system failures (recovery might redo the command). If a get is called for a replica server that is done then a rmi connection exception string is just shown to the client (should probably change this) however the client or coordinator does not crash.

Test cases)

* Failures were imitated by putting sleep commands in certain places in the code and terminating a coordinator or replica server.
* In a real world application where the program connects to an external database these test cases would most likely lead to connection leaks. So, it would be necessary for the replicas to tell the database to terminate any of its connections it lost on recovery.

1) A replica fails before recieving a request for a vote from the coordinator.

Implemention: Have one of the replicas not running when issuing a vote request from the coordinator or have the coordinator sleep when sending it to one of the replicas and then terminate that replica.

Immediate result: Coordinator logs an abort and tells the coordinators it talked to to abort the transaction.

On recovery: no recovery done here.

2) A replica fails after receiving a vote request from the coordinator but before sending out its reply.

Implemention: Have one of the replicas sleep before sending its vote to the coordinator and then shut the replica down while it is sleeping.

Immediate result: Coordinator logs an abort due to rmi connection being lost from the shut down replica during vote request method (code should be changed to implement timeout and allow replying to the coordinator on recovery) and tells the coordinators it talked to to abort the transaction.

On recovery: When the replica comes back up it will check its log history. If it finds a vote to abort then it does nothing but if it finds a vote to commit it will ask the coordinator the result of the transaction. Since the coordinator logged an abort the replica will also log it and take no further action (sqlite jdbc connections are gone and the pending transaction is aborted when the jvm shuts down).

3) Coordinator fails while recieving responses to vote requests from the replicas.

Implemention: Have the main coordinator sleep before sending a request for a vote to one of the replicas and then shut the coordinator down while it is sleeping.

Immediate result: The current transaction is pending for some of the replica servers and is added to a list of active transactions. The replica servers will also hold locks for these transactions. However, a scheduled task that runs every 3 minutes will check with the coordinator or its peers the statuses of all active transactions. If an abort is found from one of its peers then it can safely abort and release the associated resources.

On recovery: When the coordinator comes back up it checks its log and sees that the transaction is active but doesn't have an outcome. The default behavior for this situation is to abort so the coordinator logs an abort and tells every replica (even the ones that might not have received the request to vote) to abort this transaction.

4) Coordinator fails after receiving all vote requests

Implemention: Have main coordinator sleep after receiving all the responses from the replicas and then shut the coordinator down while it is sleeping.

Immediate result: Replicas will have active transactions that may not be able to proceed like in the scenario before only if all replicas voted to commit. Otherwise, they should discover that one of them aborted during their scheduled task that checks on active transactions.

On recovery: If a global abort or global commit is not logged then the recovery activity from scenario 3 is also done here (this could be made smarter by logging each individual vote request and using them to decide on an outcome). Otherwise, the coordinator will find the outcome of the vote in the log and send that to all the replica servers and log it as complete.

5) Replica server crashes right before receiving the outcome of a vote or the message drops when the coordinator sends out the outcome of the vote to all the participants

Implementation: Make the coordinator sleep before sending the outcome to one of the participants and then make that participant crash while the coordinator is sleeping.

Immediate result: The other participants either commit, abort, or are still waiting on the vote outcome. For the crashed participant it is auto aborted since it wasn't committed and the sqlite database goes down when the jvm goes down.

On recovery: The crashed participant will see that the transaction in the log does not have an outcome and will try to ask first the coordinators and then its peers (if the coordinators are down) for the result of the transaction. If any of them responds that the transaction is active the crashed partcipant will redo the update and wait for an outcome, if it was commited the crashed participant will redo the update, commit it, and then log the outcome, and if it was aborted the crashed participant just logs the abort since it was already aborted when the jvm was stopped.

6) Concurrent updates and get requests from different clients

Implementation: run commands or the test scripts from multiple clients concurrently.

Immediate results: concurrent updates to the same keys are rejected but concurrent updates for different keys are accepted.

7) Tested backup coordinator on n0001

Implementation: n0000 coordinator fails while handling commands.

Results: n0001 just takes n0000’s spot.

I chose these test cases because they were the failure and recovery situations I found when diagramming.