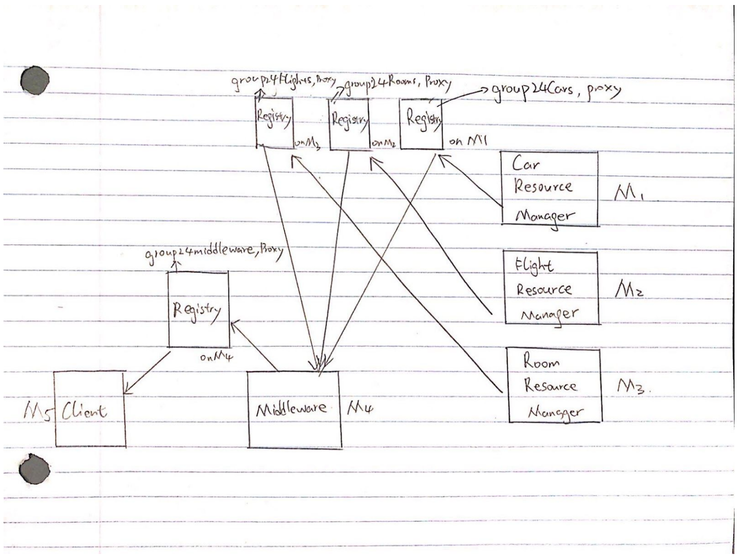
Comp 512

TONGYOU YANG 260669495
KEYU XUAN 260676756

The general design and architecture of the system implemented:
In milestone 1, we implemented RMI and TCP. For milestone 2 and 3, we used RMI architecture and there are few important entities: resource manager, transaction manager, lock manager, and middleware. Milestone 3 is basically based on milestone 2 and we add the data shadow for logging and recovery and 2pc to solve the consensus problem.

How each of the individual features of the system is implemented:

1.distribution and communication:



We have three servers running on three machines to represent car resource manager, flight resource manager, and hotel resource manager. For each server, we will first create its corresponding proxy resource manager and registry, and then register that proxy resource manager inside that registry with a name(which are all the codes given in RMI Resource Manager already).

In RMIMiddleware class, we first set up its own registry and register its proxy resource manager inside that registry, which then can be used by clients later when the client needs to perform an action. RMIMiddleware also gets proxies for all resource managers (car resource manager, flight resource manager, and hotel resource manager) by looking up in the registry using previous bounded names and later it will use these proxies to call the remote method. For all the implementations of methods from IResourceManager interface inside RMIMiddleware class, we are actually delegating each method call to real resource manager by using its corresponding proxy resource manager. For example, When a client sends a call to reserve rooms, flights or cars, middleware will just use the corresponding proxy resource manager to direct these calls to the method in the real resource manager.

But if a client types in any request about customers, RMImiddleware handles it by sending this call to all three resource managers to synchronize information about customers in all three real resource managers (car resource manager, flight resource manager, and hotel resource manager).

2.transactions and locking

For the transactions, middleware class, transaction manager class and resource manager class are involved:

In the Middleware:
We first create a transaction manager instance, then upon every reservation service call, we use the transaction manager instance to check if the resource is available. If the resource is available, then call the corresponding resource manager, otherwise return false to the client. When middleware is using the transaction manager to check the resource, it also passes the parameters to the transaction manager for later abort method call use.

For the start method call, middleware passes itself to the start method in the Transaction manager, which sets the timer for this transaction using passed middleware object and returns a transaction id to middleware.

For the commit method call, middleware simply calls the commit method in transaction manager with the transaction id.

For the abort method call, middleware calls the abort method in transaction manager, which returns a list of actions for middleware to reverse. Then abort method traverse through this returned list and call the corresponding reverse function(resides in resource manager) to reverse the action.

For the shutdown method call, middleware calls the getKeys method in transaction manager, which returns a set of transaction id to middleware. Then shutdown method call in middleware loops through the return set and call abort method on each transaction id. After aborting all transactions, shutdown methods are called on each resource manager and sets a timer to shut down itself.

In the Transaction Manager:

Each transaction manager has a transaction id counter, a lock manager object and a history hashmap, which stores all actions for all transactions.

In the checkResource method, history hashmap is updated whenever Lock method returns true, it adds that action to its history hashmap.

updateHistory method is called by reserve methods in middleware with two parameters which are given by reserve methods(passes transaction id and before images to the updateHistory method). Then updateHistory method updates the history hashmap using new arguments.

Commit method calls UnlockAll method with the passed xid, then it clears the xid entry in history hashmap.

Abort method has one parameter(transaction id) which is used to retrieve the corresponding action list and return this list to the called method.

We also created an inner class called Timethread which implement the thread class, it is used to realize the time-to-live mechanism.

In the Resource Manager:

We added methods like removeFlight, removeRoom and removeCar to the IResourceManager interface. The structures of these three methods are basically the same. We first check whether the item we want to delete exists or not, if it doesn't exist, we simply return false. If it exists and the existing number is more than the number we want to delete, we update the count and overwrite the old stored data.

We also added a method called unReservedItem, which is an opposite method compared with the reserve method.

For the locking part, it is mainly implemented in the lock manager class (two to-dos):

For the first todo in lockManager class, we first create a new TransactionLockObject and DataLockObject using given parameters, then remove read lock on these two objects with the same parameters from lockTable and add write lock on newly created objects (xLockObject and dataLockObject) to the lockTable.

For the second to do in lockManager class, we first loop through and find the one with the same xid, then check whether l_dataLockObject has read lock or write lock. If l_dataLockObject has read lock, we check every DataLockObject that holds this resource in the lockTable, then see if this DataLockObject has a write lock. If there is a DataLockObject that holds a write lock, we can determine that there is a lock conflict. If l_dataLockObject's lock type is write, we throw a redundant write lock exception.

3.data shadowing
Each ResourceManager has two files: one committed version and one for transactions in-progress. For this part, we created a MasterRecord class, which stores a field called activecopy, this field is used to keep track of which file is the committed version and it is set to opposite every time the program successfully commits in the resource manager. In the ResourceManager class, we create a MasterRecord instance called master. When we want to save m_data to disk in ResourceManager's commit method, the opposite of activecopy is used to generate or look for the correct file name. After saving m_data to the shadowing file, the master object is first saved to disk, then its activecopy field is set to the opposite. When

resource manager wants to restore m_data every time it boots up, it will first read the saved master record instance, then look for the correct committed file using the restored activecopy field to restore m_data.

4.logging and recovery(2pc)
To achieve 2pc, we need to have one coordinator(transaction manager) and several participants(resource manager). The coordinator and participants may crash at any time points.

When the user wants to do the commit, the coordinator will first write the “start 2pc ” log in the log file for the transaction id and save it to disk.

If the coordinator crashes at this time point, it will just abort this whole transaction. If at this time point coordinator has not crashed, it will call the prepare method of three resource managers. There are several situations:

#1: coordinator crashed after sending vote request(call the prepare method of resource manager) and before receiving any replies. Then the coordinator will simply call self destruct method

#2: coordinator crashed after sending vote request and before receiving any replies. We used a container to pretend we have not received any replies(but actually we already called the prepare method of resource manager). Then the coordinator will simply call self destruct method

#3: coordinator crashed after receiving some replies but not all. We just append the “some replied” into the log file of this transaction, then the coordinator will simply call self destruct method

#4:coordinator crashes after receiving all replies but before deciding. We append an “after replies before decision” record into the log file of this transaction. Then the coordinator will simply call self destruct method

#5: if all participants vote yes and coordinator crashes after deciding but before sending decision. We append a “before commit all vote yes” into the log file of this transaction.Then the coordinator will simply call self destruct method. Similarly, if one of the participants vote no and coordinator crashes after deciding but before sending decision. We append a “before commit some vote no” into the log file of this transaction.Then the coordinator will simply call self destruct method.

#6: if all participants vote yes and coordinator crashes after sending some but not all decisions, we append a “some committed” into the log file of this transaction.Then the coordinator will simply call self destruct method. Similarly, if one of the participants vote no and coordinator crashes after sending some but not all decisions. We append a “some aborted” into the log file of this transaction.Then the coordinator will simply call self destruct method.

#7: if all participants vote yes and coordinator crashes after having sent all decisions, we append an “after commit” into the log file of this transaction.Then the coordinator will simply call self destruct method. Similarly, if one of the participants vote no and coordinator crashes after having sent all decisions. We append a “after abort” into the log file of this transaction.Then the coordinator will simply call self destruct method.

#8: recovery of the coordinator:
First of all, in the main method of transaction manager class, we first need to load the history from disk. Then we call the recoverMiddleware method:

if the number of log is 1 (“START_2PC_LOG”) or 2 (“START_2PC_LOG”,”SOME_REPLIED”) or 3(“START_2PC_LOG”,”SOME_REPLIED”,”AFTER_REPLIES_BEFORE_DECISION”) or 4(“START_2PC_LOG”,”SOME_REPLIED”,”AFTER_REPLIES_BEFORE_DECISION”,”BEFORE_COMMIT_ALL_VOTE_YES”) (crash case 1-5), we will simply resend the vote request.

If the number of log is 5(“START_2PC_LOG”,”SOME_REPLIED”,”AFTER_REPLIES_BEFORE_DECISION”,”BEFORE_COMMIT_ALL_VOTE_YES”, “SOME_COMMITTED”) (crash case 6), resend commit.

If the number of log is 6(“START_2PC_LOG”,”SOME_REPLIED”,”AFTER_REPLIES_BEFORE_DECISION”,”BEFORE_COMMIT_ALL_VOTE_YES”, “SOME_COMMITTED”, “AFTER_COMMIT”) (crash case 7), do nothing.

In the **resource manager**, there are five scenarios to crash:

#1: Crash after receive vote request but before sending the answer. In this case, there is no log in the log file. In the recover protocol, we just abort the transaction.

#2: Crash after deciding which answer to send (yes/no). In this case, there is only one log record in the log file, which is BEFORE_YES/BEFORE_NO. Thus, resource manager has decided which answer to send but not yet sent the decision before resource manager crashes. The coordinator would find out that resource manager is down in this case, and it will continuously try to connect to middleware and send the vote request until it times out. Resource manager, on the other hand, will wait indefinitely until it receives the vote request from the coordinator.

#3: Crash after sending the answer. In this case, we call selfDestruct method, which exits the program after 500 milliseconds after prepare method returns. The log file for the resource manager would have two log entries (BEFORE_YES/BEFORE_NO and AFTER_YES/AFTER_NO). When the resource manager tries to recover, it would ask middleware for the vote result from all resource managers. If all the results are yes, it would redo all the operations for this transaction. If some results are no, it would just save the m_data to the disk and delete the history file and log file.

#4: Crash after receiving the decision but before committing/aborting. In this case, the program is crashed at the beginning of the commit/abort method. Thus the log file for the resource manager would have two log entries (BEFORE_YES/BEFORE_NO and AFTER_YES/AFTER_NO). When the resource manager tries to recover, it would ask middleware for the vote result from all resource managers. If all the results are yes, it would redo all the operations for this transaction. If some results are no, it would just save the m_data to the disk and delete history file and log file.

#5: Crash during recovery. In this case, we set the crash after we have done the recovery and written the log into our log file. After the recovery, we write a special log, “RECOVER_COMPLETED” into our log file. Thus, on the restart, the resource manager will check if the log file contains this record. If there is no “RECOVER_COMPLETED” found, normal recover protocol will be executed. If “RECOVER_COMPLETED” is found, resource manager will just skip the recover protocol and continue.

Special features your implementation
For this milestone, we implemented the full recovery of the resource manager and coordinator.

Problems that you encountered
#1: When we were doing the recovery for resource manager, we are supposed to read the commited version of the shadow file. But at first we made a mistake and read the working version of the shadow file and got null pointer exception error

#2: When we were doing the second crash case for the coordinator(coordinator crashed after sending vote request and before receiving any replies), we used a container to pretend we have not received any replies yet.

#3: In milestone 2, we only need transaction manager to store all the histories of all the transactions. But in milestone 3, we also need three resource managers to have the same histories for the recovery purpose. So we add a field in the resource manager class to store the history, and it will be updated by the transaction manager

How you test the system for correctness

For this milestone, we tested the system using every crash mode mentioned in the assignment. Also we26 also tested the recovery of each crash mode in the resource manager and middleware.