**Distributed Systems CS565**

***Write up – Transaction Server P3.2***

*Shreya Balasubramanya Raju*

*Sai Greeshmanth C*

*Goutham Kumar J*

* **Brief Overview of our Project Implementation –**

***Transaction management****;* refers to the tasks of processing multiple transactions issued by various clients of a database server in such a way that the ACID contract can be fulfilled, that is, the properties of *atomicity*, *consistency preservation*, *isolation*, and *durability* of each individual transaction can be guaranteed. Transaction management is generally understood as requiring serializability-based concurrency control as well as recovery from failures. Concurrency control is the task of scheduling transactions such that their serializability can be guaranteed.

***A transaction server***; is a specialized type of server that manages the operations of software-based transactions or transaction processing. It manages application and database transactions on a network or Internet, within a distributed computing environment.

***A transaction client***; uses some server proxy objects that mimic the presence of the server on the client-side and provide an interface that the client talks to. These objects, from a client’s perspective, represent the transactions. Overall, the number of transactions to be run on the server is configurable. Clients will accept the connectivity information of the server as a configurable property.

***Deadlocks***; the idea is to check for the *potential* of deadlocks occurring right at the moment when a transaction tries to acquire a lock. In this situation, two things may happen. Either the lock is being granted and we are fine. Alternatively, the lock request is denied because it conflicts with a lock that has already been set by another transaction. That means that the transaction requesting the lock needs to wait for the other transaction(s) to release the lock. And this is the critical moment where a circle of transactions waiting for each other may build. We check all the locks that the current transaction is holding. If there is any other transaction waiting for any of those locks we ABORT (“close Transaction”) the current transaction and thus break a cycle that may be in the process of forming. A simple, yet efficient measure to prevent deadlocks from happening.

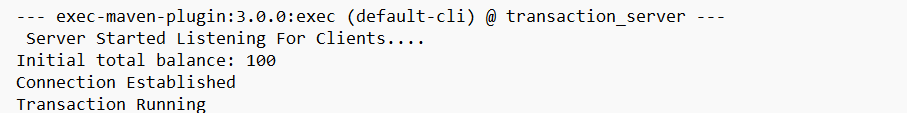
***Transaction Operations***; conceptually transaction server can support any number of read/write operations on its data objects in any order. All transactions hitting the server will only ever do the same set of operations that each and every transaction is comprised of: an arbitrarily chosen dollar amount is withdrawn from an arbitrarily chosen account and deposited onto another, arbitrarily chosen account. Transactional server for 10 accounts with an initial balance of $10 hit by 20 transactions. Running it once with locking in place and another time without a lock.

***Transaction management with Locking***; each transaction requests locks of different types on the resources, such as rows, pages, or tables, on which the transaction is dependent. The locks block other transactions from modifying the resources in a way that would cause problems for the transaction requesting the lock. Each transaction frees its locks when it no longer has a dependency on the locked resources.

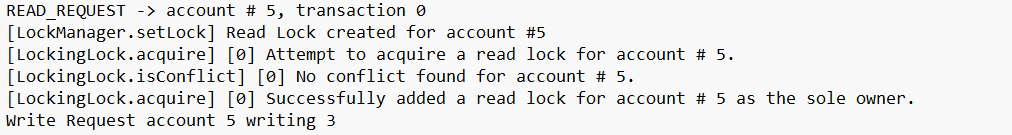
* ***Working Mechanism-***

1. **LOCKING**

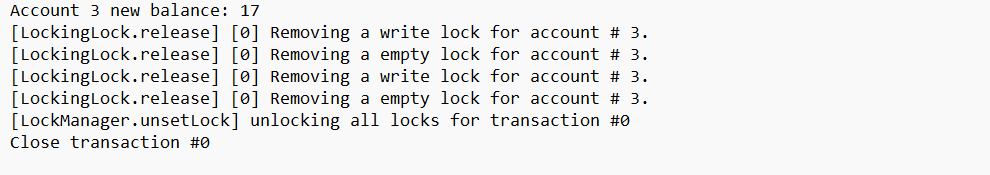
* *SERVER-SIDE:*



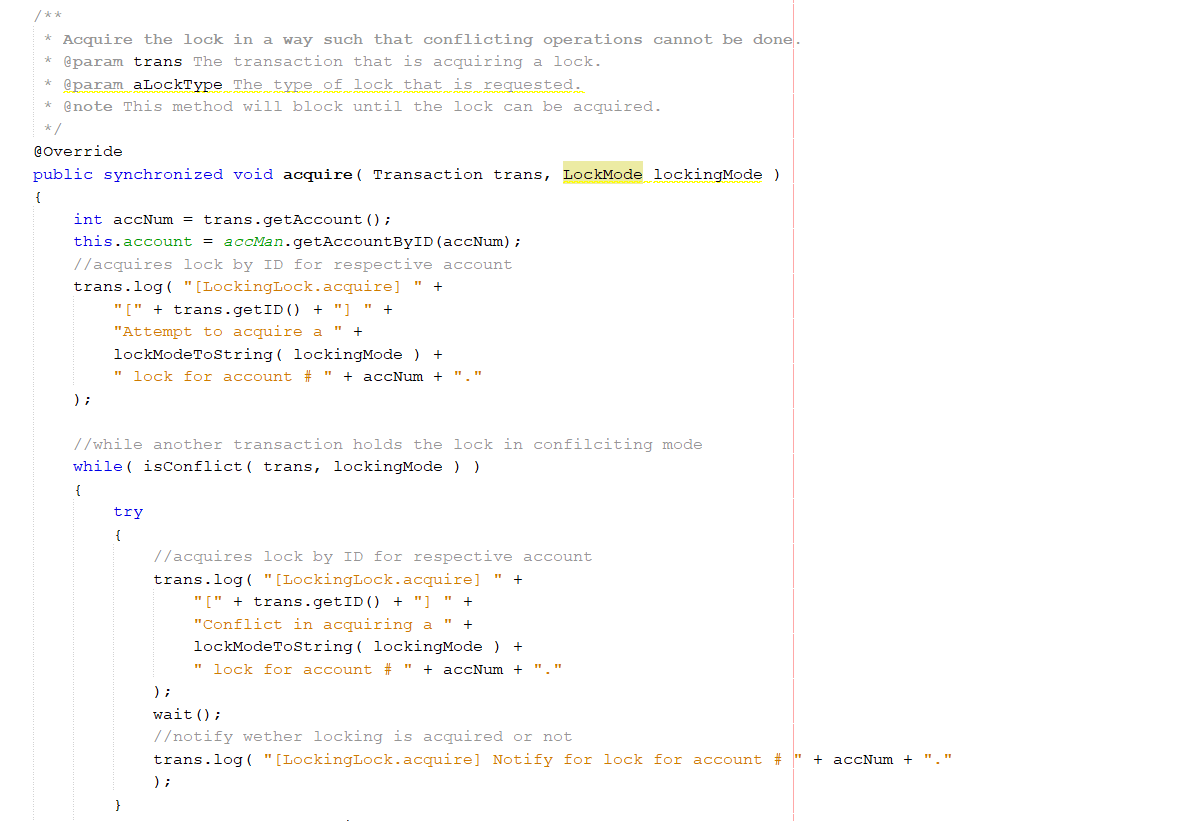
* The server is up and running and waiting for the client-side to respond. The initial balance specified is $100, the above shows that the connection is successfully established, and the transaction starts running.
* A Transaction opens and starts read and write requests, first a read request is started, and a lock is acquired if there are no conflicts similar to the write requests, then we credit or debit money using write requests.
* Let us take Transaction 0 as a sample and see what goes on.
* Transaction 0:

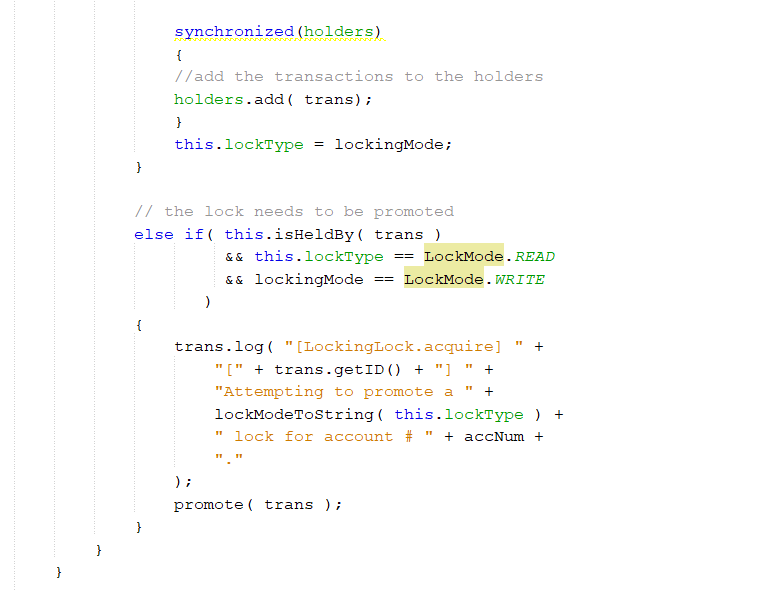


* At first, a read lock is created for account 5 by LockManager and an attempt is made to acquire the lock for account 5, since there are no conflicts that are verified by Locking.isConflict, we can add a read lock for account 5.
* After read lock, write request is called to credit or withdraw the money, we can see 3 dollars being credited in the above pic. We can see setlock, acquiring the lock for account 5.
* After the transfers are done within accounts, conflicts will be checked between transactions, if there aren’t any, then all the locks are removed and closes the transaction.
* Similarly read, write requests, locks are processed for every transaction and validated with other transactions in the end all transactions are closed. We can see all the locks are being removed for account 3 and the new balance is updated for the respective account.

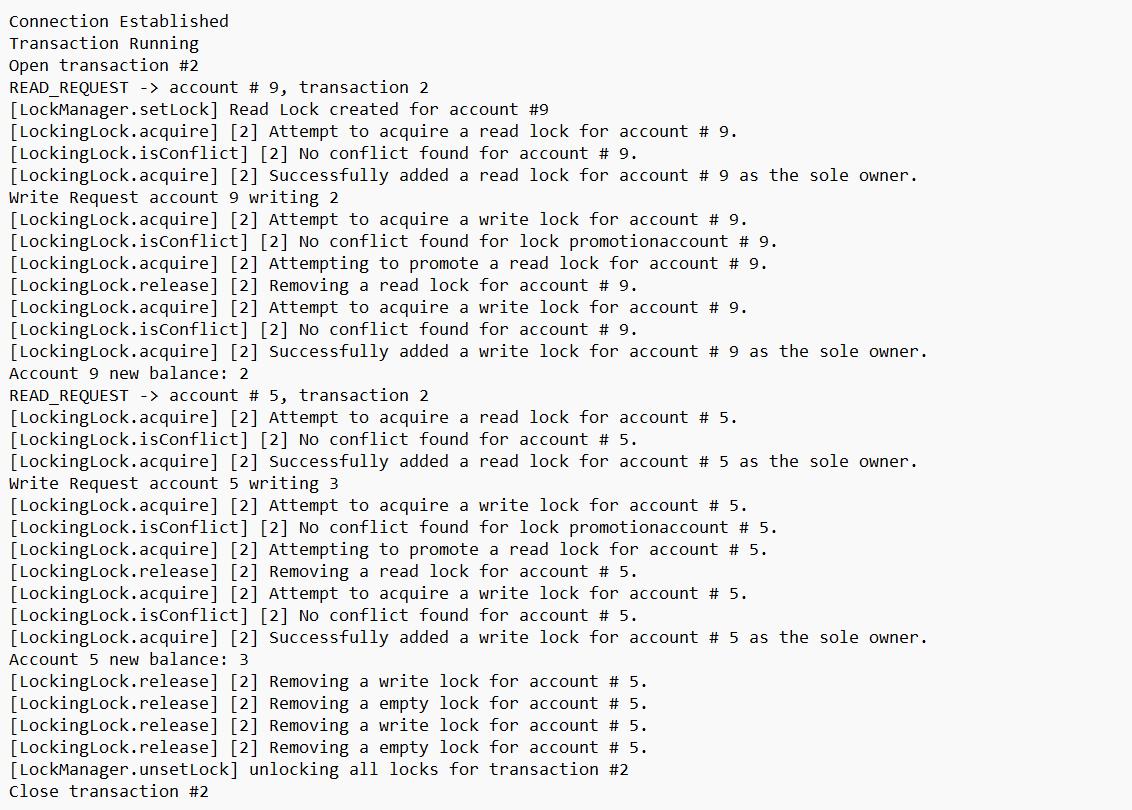


* We coded in such a way that the conflicting operations will not occur in both the lock and non-locking, with the help of this function, the conflicting operations cannot be done.
* Here we get the account number and the account ID, in this method we check whether a conflict is occurring for the respective transaction and accordingly the locking mode is enabled.
* It executes in such a way that even if the conflict occurs it doesn’t make any issues with the transaction. The first transaction to occur executes first, and since locking is made for every account there won’t be a problem with mismatching the accounts while running.
* What happens in this function is, if a conflict raises between read-write operations, the conflicting transactions are managed in such a manner that while one transaction is being executed the other transaction is placed in a holder wherein it waits until the other transaction is completed. This particular transaction is synchronized with the ongoing transaction later by using the synchronized void release function. The transactions are removed from the holders and the returns the empty holders once this process is completed.





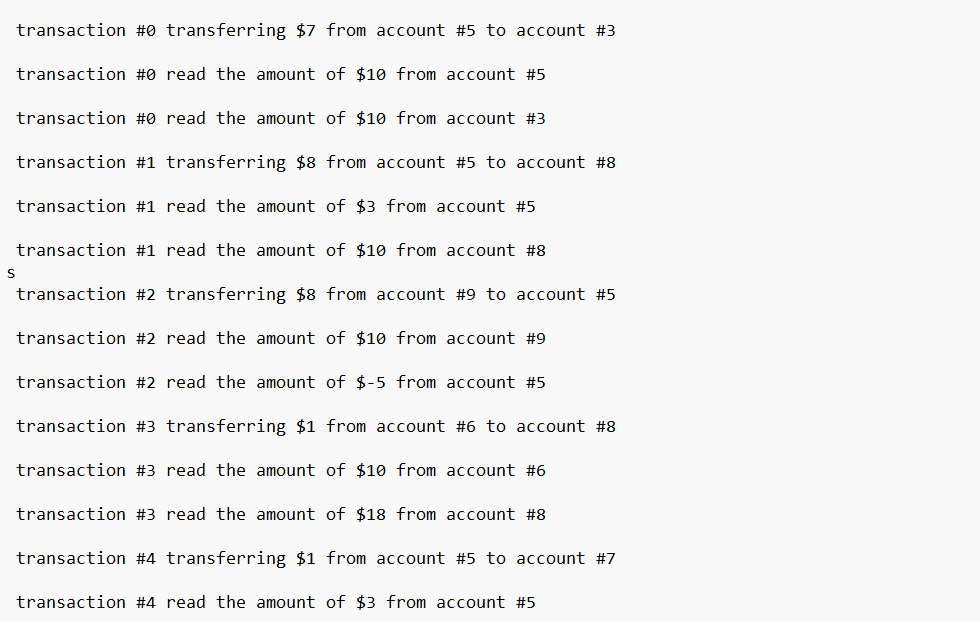
* Similarly for transaction 2 this the process that occurs:



* First a read lock is acquired to an account and attempts a write lock as well to make the transfer of money in between accounts. We can see “write request account 9 writing 2” , updating the account 9 balance to $2 and later account 5 is started, and the same process is repeated with locking. New balance is updated to $3 for account 5.
* After all the transactions are done, the Ending total balance is checked and should be same as the initial Total balance, so therefore no loss is obtained.
* The final balances of the accounts are also updated.

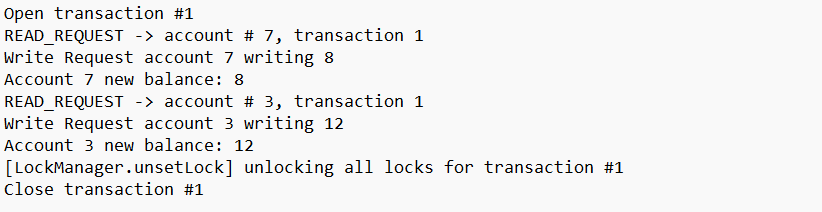


* *CLIENT-SIDE:*
* Now on the client-side, the transfer of amounts between accounts occurs, for example;

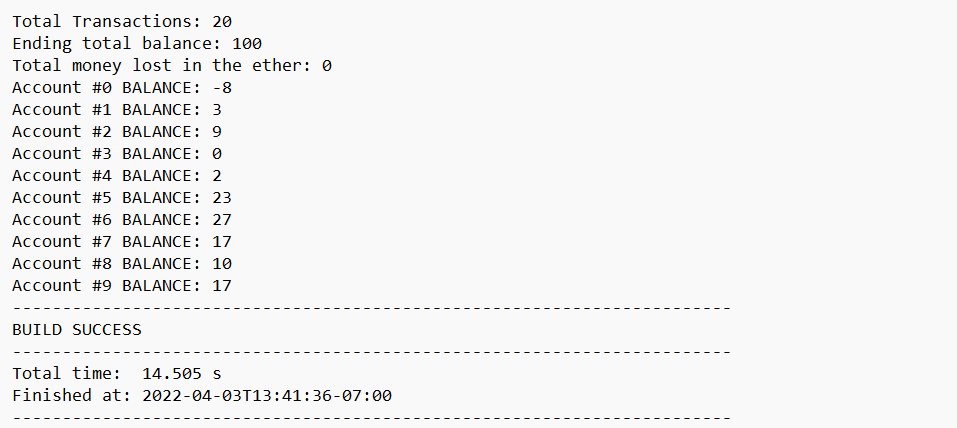


1. **NON LOCKING:**

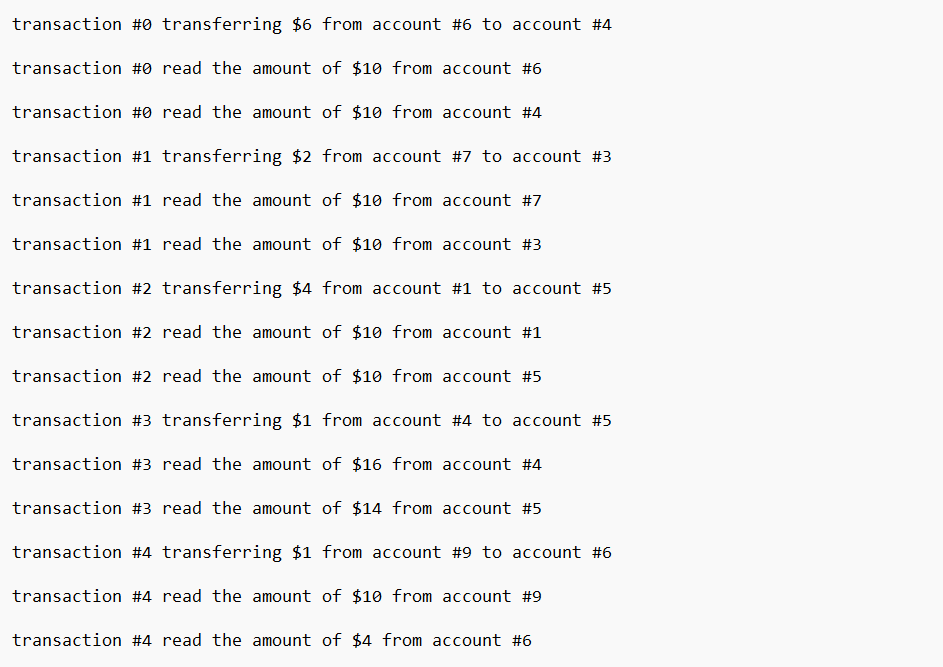
* *Server Side:*
* In case of no Locks, the transactions are not locked so only read-write requests are implemented, lock manager will unset all the locks.
* Opens a transaction #1



* As we can see it opens a transaction and read request with account 7 in transaction 1 and using the write request 8 dollars amount is added to the account 7 and balance is updated and the new balance is printed, similarly, for account 3, 12 dollars are added using the write request.
* Similarly, many transfers are done within a transaction and the final balance is updated for every account
* In the end, the Lock manager unset all the locks in the transaction.
* This goes the same for all the transactions, the final balance is also checked in the end which should match the initial balance.



* *Client-Side:*
* Transferring of the money between accounts is done on the client-side.



* Only the transferring and the read amounts are printed on the client-side and get updated as final balances on the server-side.