Northeastern University – Silicon Valley

CS 6650 Scalable Dist Systems

Homework #4 [100 points]

Name: Nisarg Patel
Email: patel.nisargs@northeastern.edu

***INSTRUCTIONS:* Please provide clear explanations in your own sentences, directly answering the question, demonstrating your understanding of the question and its solution, in depth, with sufficient detail. Submit your solutions [PDF preferred]. Include your full name. Do not email the solutions.**

Answer the following questions using explanation and diagrams as needed. No implementation needed. Study **Chapter 16 from** Coulouris Book

1. 16.1 [10 points]

Ans:

(i) the server can reply immediately, telling the client to try again later

Advantages: The client(worker) does not have to wait for master to add a task in the bag.

Drawbacks: Client have to make multiple calls to the master if the bag is empty. This method can lead to starvation(unfairness) in the waiting worker as the request from other workers may be processed before the worker sends request again.

(ii) make the server operation (and therefore the client) wait until a task becomes available.

Advantages: Worker have to make a single call to the master to get a task. Fairness can be achieved using this.

Disadvantages: If no task is available to take, the worker has to wait till master adds a task in the bag.

(iii) use callbacks.

Advantages: Worker does not have to make multiple calls to get a task. The server can call notify to the worker when a task is available.

Disadvantages: The stack handling becomes complex for larger and more complex distributed systems.

---

2. 16.2 [10 points]

Ans:

If T -> U

Interleaving – A	Interleaving – B	Interleaving – C
T1: x = read(j)	T1: x = read(j)	T1: x = read(j)
T2: y = read(i)	U1: x = read(k)	T2: y = read(i)
U1: x = read(k)	T2: y = read(i)	T3: write(j, 44)
T3: write(j, 44)	T3: write(j, 44)	U1: x = read(k)
T4: write(i, 33)	T4: write(i, 33)	T4: write(i, 33)
U2: write(i, 55)	U2: write(i, 55)	U2: write(i, 55)
U3: y = read(j)	U3: y = read(j)	U3: y = read(j)
U4: write(k, 66)	U4: write(k, 66)	U4: write(k, 66)

Final values:
x = initial value of a_k
y = 44
a_i = 55
a_j = 44
a_k = 66

If U -> T

Interleaving – D	Interleaving – E	Interleaving – F
U1: x = read(k)	U1: x = read(k)	U1: x = read(k)
T1: x = read(j)	U2: write(i, 55)	U2: write(i, 55)
U2: write(i, 55)	U3: y = read(j)	U3: y = read(j)
U3: y = read(j)	T1: x = read(j)	T1: x = read(j)
T2: y = read(i)	T2: y = read(i)	T2: y = read(i)
U4: write(k, 66)	U4: write(k, 66)	T3: write(j, 44)
T3: write(j, 44)	T3: write(j, 44)	T4: write(i, 33)
T4: write(i, 33)	T4: write(i, 33)	U4: write(k, 66)

Final values:
x = initial value of a_j
y = 55
a_i = 33
a_j = 44
a_k = 66

---

3. Concurrency Control and Java Apps
    a. Explain the 3 different concurrency control methods (Ch 16 Coulouris Book – Locks based, Optimistic and Time-stamp Ordering) brifely. Compare and contrast how they achieve concurrency control. [10 points]

Ans:

Locks: In locks-based concurrency control, the object that is used during a transaction is locked by the server to perform operations and other server requests that tries to access that object must wait till the object is unlocked. In strict two-phase locking, the locks are held when the object is first accessed during the transaction and released when the transaction either commits or aborts.

Optimistic Concurrency Control: It assumes that the likelihood of two or more transactions that tries to access the same object is rare (hence optimistic). It consists of 3 phases:
(i) Working Phase: Each transaction proceeds without any wait and stores the updates.
(ii) Validation Phase: When transaction ends, the operations are validated whether it conflicts with other transactions or not. If conflict, then a resolution has to be made and some transaction has to be aborted in most of the cases.
(iii) Update Phase: The changes made during working phase to all the objects are made permanent.

Timestamp Ordering: During start of transaction, it is assigned a timestamp. It follows the rule that the write operation of a timestamped transaction is valid only if the previous read or write operation on that object is done by an earlier timestamped transaction. Similarly, read operation for a timestamped transaction is valid only if the previous write operation is done by an earlier timestamped transaction. If the transaction is failing the rule in any of the object, then it is aborted immediately.

Comparing the three concurrency control methods:
The two-phase locking and the timestamp ordering are pessimistic approaches and detects the conflicts as soon as the object is accessed. The optimistic concurrency on the other hand is optimistic in the way that the conflicts are detected close to the commit time.

The two-phase locking makes the transaction wait till the conflicting object is available to access. The timestamp ordering and the optimistic concurrency aborts the transaction if conflict is detected.

The two phase locking method includes overhead of lock object. Furthermore, it includes waiting of processes, it is susceptible to deadlocks. Thus, deadlock prevention measures like timeout and deadlock detection also increasing time overhead.

Optimistic concurrency control and timestamp ordering are both time and space efficient since they abort immediately if it has to, but can result in starvation. Furthermore, if there are long transactions, and it fails, then substantial amount of work has to be repeated (more in case of optimistic as all transaction is processed before validation phase).

---

b. **Java Concurrency In Practice** by Brian Goetz is the practice manual for Dist Systems. Study Chapter 6 (Task Execution) which is a small chapter.
Identify 5 different programming devices (e. g. Tasks, Exec Framework) used in Java for Task Execution. Then, briefly explain how you might use these to implement a basic Locks based concurrency control in a simple Web App. [15 points]

Ans: The following are some of the different programming devices used for Task Execution in Java:
(i) Tasks in Threads:
Instead of executing tasks sequentially, a thread can be created for servicing each request. We can create one lock for each object. Inside the run() method of Runnable task, we can call synchronized(LOCK_OBJECT) for every object used by the thread. This would help for basic concurrency control in such thread based task executing web applications.
(ii) Executor Framework and ExecutorService
The thread-based task execution has some limitations, like creating multiple threads can result in threads sitting idle if not enough CPUs are present and worse, can make applications run out of memory. To overcome these limitations, Executor framework can be used. It is contained by java.utils.concurrent and it's interface is as follows:

```
public interface Executor {
    void execute(Runnable command);
}
```

It is primarily based on producer-consumer pattern, and provides a decoupling between task submission and task execution.

To keep track of the state of the Executor, important for running applications to handle shutdowns and to keep track of completed tasks, ExecutorService interface can be used. The executor service lifecycle keeps track of three states: running, shut down and terminated.

The lock based implementation described in thread tasks can also be applied here as well since synchronized(LOCK_OBJECT) is conducted in Runnable for executor framework as well.

(iii) ThreadPool

A thread pool manages multiple worker threads. In this framework, if one thread completes a task, it waits for another task to be assigned by the thread pool. Thus, it helps to reduce the overhead cost of creating new threads. A thread pool can be created by calling one of the following static methods in Executor.

newFixedThreadPool: Fixed sized thread pool.
newCachedThreadPool: Dynamic sized thread pool based on demand.
newSingleThreadExecutor: Single worker thread.
newScheduledThreadPool: Fixed size, but supports delayed as well as periodic execution.

All of them returns an ExecutorService, so lock-based concurrency control for executors can be applied to thread pools as well.

(iv) Result-Bearing Tasks (Callable and Future)

Many of the tasks require a result to be returned to the Caller for further processing. Runnable interface does not return any value. But it can be achieved by the use of Callable interface. Since tasks are executed asynchronously, to know the lifecycle of such tasks can be done using Future interface. The 'get' method of Future returns immediately if the task is completed or blocks if the task is not yet completed. The executor.submit() method takes a Callable as an input and provides a Future as a return value.

Similar to Runnable, the lock-based concurrency control can be achieved by putting synchronized(LOCK_OBJECT) inside 'call' method in Callable.

(v) Completion Service

To continuously retrieve the result of different tasks and perform operations continuously on retrieved tasks, the CompletionService framework can be used. It can be considered as a framework that combines the functionalities of an Executor and a BlockingQueue. The completionService.take() method polls a completed task from the queue of completed tasks. Since completionService also uses Callable, the lock based concurrency control for Callable can be used in this framework.

---

4. Refer to Oracle Database Administrator's Guide page 29-23 [15 points]

Transaction Processing in a Distributed System

https://docs.oracle.com/html/E25494_01/ds_concepts004.htm But get the PDF, it is easier to read.
What are distributed transactions in Oracle DB? How are the different from Remote Transactions? Give examples. How does Oracle DB use Naming service and 2-Phase Atomic Commit Protocols to manage distributed transactions?

Ans:

In Oracle Database, a query consist of SQL instructions. The distributed database is stored in many nodes. A remote query or update is a query or update that references one or more remote tables that are stored in a single remote node.

Remote Transaction:
A remote transaction consists of one or more such remote statements that are stored in a single remote node. An example of remote transaction:

```
UPDATE cs@khoury.northeastern.edu
SET location = 'BOSTON'
WHERE id = ‘patel.nisargs’;
UPDATE ai@khoury.northeastern.edu
SET location = ‘SILICON VALLEY’
WHERE id = ‘allen.tikker’
COMMIT;
```

Distributed Transaction:
Whereas, a distributed transaction consists of one or more such remote statements that are stored in two or more remote nodes. A example of a distributed transaction:

```
UPDATE cs@khoury.northeastern.edu
SET location = 'BOSTON'
WHERE id = ‘patel.nisargs’;
UPDATE ict@daiict.ac.in
SET currentLocation = ‘BOSTON’
WHERE id = 201501134;
COMMIT;
```

Two Phase Commit Protocol:
To ensure the data integrity of the global database in all the nodes of the distributed database, all the nodes should perform same action: either all commit or all abort the transaction. To achieve this, Oracle DB uses the two-phase commit protocol. It consists of the following phases:

(i) Prepare Phase: In this phase, the initiating node, known as the global coordinator, all the nodes in the distributed transaction that are referenced, are told to prepare themselves for the commit by recording the information in a redo log, such that it can either commit or abort the transaction afterwards. They also place a distributed lock on the modified table to prevent further operations on it. Each of such referenced node then responds back to the global coordinator. The responses can be one of: Prepared: Prepared to handle the request from the global coordinator(Commit or Abort), Read-Only (No updates made, so no preparation required) or Abort(Cannot commit the transaction, so need to abort it).

(ii) Commit Phase: After all the nodes referenced in the distributed transaction, except the commit point site has responded as Prepared to the global coordinator, the commit phase begins.
First the commit point site commits. Afterwards all the other referenced nodes are asked to commit. Each of the nodes then responds back that it has committed. At this point, the global database is consistent.

(iii) Forget Phase: After all the nodes have committed, the transaction can be forgetted by the commit point and the global coordinator.

Naming service:
A global object name which is used to specify a database link consists of 3 components:

	Object name	Database Name	Domain
Eg:	cs	khoury	northeastern.edu

It can be referenced as cs@khoury.northeastern.edu
A database link consists of the Database name and the Domain. The database searches for the database links in the following order:
(i) Private Database Links present in the schema of the caller.
(ii) Public Database Links present in the local database.
(iii) Global Database Links present in the server directory.

If a partial database link is specified, then database uses the initialization parameters to complete the global database name lookup. If no database link is specified, then the database assumes that the reference is for a local database.

---

5. Study Alibaba Fescar note. How does Fescar manage distributed transactions? What protocols are used? Describe in detail. How does it avoid Deadlock? [20 points]
Ans:

Fescar is an open-source version of Alibaba's GTS solution for distributed transaction in microservices. In Fescar, a distributed transaction is considered as a global transaction with several branch transactions. The following components are used for the process of distributed transaction:
(i) Transaction Coordinator (TC): It maintains the current state of all the global transactions. It also coordinates the commit or abort(rollback) of the global transactions.
(ii) Transaction Manager (TM): It opens a global transaction and makes decision of whether to commit or rollback a transaction.
(iii) Resource Manager (RM): Local to branch transactions, it registers the branch and provides it's status. It receives the instructions from transaction coordinator whether to commit or abort the branch transaction.

Fescar manages the distributed transaction by performing a variation of two-phase commit protocol. The steps of a typical two-phase protocol using TC, TM and RM are as follows:
TM starts a global transaction in TC, which creates a unique transaction ID (XID) for the global transaction. Using Microservices, the XID is shared with the RM. RM then registers it's branch to the TC and creates a copy of the updates, sending response to the TC. Based on the responses, TC initiates a global commit or rollback for the XID to the TC. TC shares the initiation with the RMs to either commit or rollback the branch transactions.

For a normal two-phase protocol, the lock is acquired for both the phases – The prepare phase and the commit phase. Fescar, however, considers that most of the transactions(at least 90%) will be committed without any issue. So, it acquires locks for only the first phase, and commits the transaction in the first phase itself. This reduces the transaction lock time and thus increasing the concurrency. In the second phase, if the result is commit, then it does nothing, else it preforms a special rollback() operation if abort is initiated.

To achieve this, the Fescar's RM (JDBC Data source proxy) is at the application side on the middleware layer. The JDBC stores the data images before and after a branch transaction and thus maintaining an update image with each XID and its corresponding undo log. Using this, the branch transaction can be committed using the update image and release the log immediately. During phase two, if the resolution is commit, then it does nothing and returns very quickly. If the resolution is global abort, then it finds the corresponding undo log of the XID and perform the rollback() operation which essentially performs the operations in the undo log.

Since the lock is held for only a single phase and is released as soon as the transaction is committed on first phase, the deadlock is avoided in the case of Fescar.

---

6. Transaction API (JTA) allows applications to perform distributed transactions. For a small e-commerce retail store, explain a design for implementation using JTA. Show the key Classes and how you use them to resolve concurrency issues and deadlock. [20 points]
Ans:
Transaction Identifier class: It is used to create a specific identifier for each transaction.

```
import javax.transaction.xa.Xid;

public class TransactionIdentifier implements Xid {

    private final int formatId;
    private final byte[] globalTransactionId;
    private final byte[] branchQualifier;

    public TransactionIdentifier(int formatId, byte[] globalTransactionId,
byte[] branchQualifier) {
        this.formatId = formatId;
        this.globalTransactionId = globalTransactionId;
        this.branchQualifier = branchQualifier;
    }

    @Override
    public int getFormatId() {
        return formatId;
    }

    @Override
    public byte[] getGlobalTransactionId() {
        return globalTransactionId;
    }
```

```java
    @Override
    public byte[] getBranchQualifier() {
        return branchQualifier;
    }
}
```

Creating a DataSource: It is used to provide a data source for transaction. This could improve concurrency by providing dynamic server details.

```java
import com.microsoft.sqlserver.jdbc.SQLServerDataSource;

import javax.sql.DataSource;
import java.sql.SQLException;

public class DataUtil {
    public static DataSource getDataSource()
            throws SQLException
    {
        SQLServerDataSource xaDS = new SQLServerDataSource();
//        xaDS.setDataSourceName("SQLServer");
        xaDS.setServerName("localhost");
        xaDS.setPortNumber(5432);
        xaDS.setPassword("1807");
        xaDS.setSelectMethod("cursor");
        return xaDS;
    }
}
```

Transaction class for Buy Method: Each of the transaction operations can be created as a class as shown below. Here is an example of a simple Buy transaction in an e-commerce store. It updates the inventory of the item in the store and creates a record of the bought item in the User.

```java
import javax.sql.XAConnection;
import javax.sql.XADataSource;
import javax.transaction.xa.XAException;
import javax.transaction.xa.XAResource;
import javax.transaction.xa.Xid;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;

import static Q6.DataUtil.getDataSource;

public class TransactionBuy {

    public static void Transaction_Buy(int user_id, int item_id, int quantity)
throws SQLException {
        XADataSource xaDS;
        XAConnection xaCon;
        XAResource xaRes;
        Xid xid;
        Connection con;
        Statement stmt;
        int ret;

        xaDS = (XADataSource) getDataSource();
        xaCon = xaDS.getXAConnection("jdbc_user", "jdbc_password");
        xaRes = xaCon.getXAResource();
        con = xaCon.getConnection();
        stmt = con.createStatement();
        xid = new TransactionIdentifier(100, new byte[]{0x01}, new
byte[]{0x02});
        try {
            xaRes.setTransactionTimeout(10);
            xaRes.start(xid, XAResource.TMNOFLAGS);
            stmt.executeUpdate(String.format("UPDATE INVENTORY SET quantity =
%d WHERE item_id = %d", quantity, item_id));
            stmt.executeUpdate(String.format("INSERT INTO USER VALUES (%d,
%d)", item_id, quantity));
            xaRes.end(xid, XAResource.TMSUCCESS);
            ret = xaRes.prepare(xid);
            if (ret == XAResource.XA_OK) {
                xaRes.commit(xid, false);
            }
        }
        catch (XAException e) {
            e.printStackTrace();
        } finally {
            stmt.close();
            con.close();
            xaCon.close();
        }
    }
}
```

Deadlock can be prevented by adding a timeout mechanism as shown above:
`xaRes.setTransactionTimeout(10);`