**Name:** Jamie Brindle

**ID:** 06352322

**Unit Code:** 63SN3201

**Unit Title:** Distributed Systems

**Course:** Bsc(hons) Computer Systems

**Level:** 3

**Title:** Assignment 2

**Tutor:** Dr Helen Hicks

**Introduction**

This assignment will deal with transaction processing. A server manages data items a*1*,a*2*,…a*n*. The server provides two operations for its clients:

*Read (i)* returns the value of a*i*

*Write (I, Value)* assigns the value to a*i*

The transactions T, U, V and W are defined as follows:

**T:** *x: = read (i); y: = read (k); write (l, 33)*

**U:** *z: = read (j); write (I, 44)*

**V:** *w: = read (k); w: = read (i); write (k, 66)*

**W:***z: = read (m); write (m, 55)*

The initial values are a*i* = 10; a*j* = 20; a*k* = 30; a*l* = 40; a*m* = 50.

x, y, z, w are variables local to the transaction

The order in which the transactions attempt to commit are **T,V,U,W**

**Part A: Optimistic Concurrency Control**

In order to achieve concurrency here, a validation technique is used, whereby the outcome of the validation affects whether or not a transaction commits or aborts. There are 3 basic rules to the validation. T*v* is the transaction being processed and T*i* is an earlier transaction which is still active. Only one transaction can be in the update or validation process at one time, in which case a transaction would have to wait until a transaction’s validation or update process has completed before validating or updating, time is represented as you go from left to write.

Validation Rules:

|  |  |  |
| --- | --- | --- |
| T*v* | T*i* | Rule |
| *write* | *read* | 1. T*i* must not read objects being written byT*v* |
| *read* | *write* | 1. T*v* must not read objects being written byT*i* |
| *write* | *write* | 1. T*i* must not write objects being written byT*v*   T*v* must not write objects being written byT*i* |

Outlined by

There are two methods of validation which include ‘backwards validation’ and ‘forwards validation’. Backwards validation looks at the transaction (T*i*) which entered the validation phase before it to determine whether or not to commit, forwards validation looks at the transaction undergoing validation with other later transactions to determine whether or not to commit .

In backwards validation as all the read operations of earlier transactions were performed before the validation part of Tv (no two transactions can be in the validation state or update stage at the same), they can’t be affected by the writes of current transactions and so rule 1 will always be satisfied. In forward validation rule 2 is satisfied as the next overlapping transactions do not write until after the current transaction has completed .

**Using Backwards Validation**



Updated Values: a*i* = 44, a*j* = 20, a*k* = 66, a*l* = 33, a*m* = 55

1. **Using Forwards Validation**



Updated Values: a*i* = 44, a*j* = 20, a*k* = 66, a*l* = 33, a*m* = 55

**Part B: Time Stamp Ordering**

**New Values:**

|  |  |
| --- | --- |
| ai = 44 | al = 33 |
| aj = 20 | am = 55 |
| ak = 66 |  |

In timestamp ordering, each transaction is given a unique ‘timestamp’ when a transaction is opened and each operation within a transaction is assigned a read timestamp and a write timestamp equal to the transaction’s timestamp that the operation is in. For example, if transaction T has a timestamp of 1 and reads object i, then i would be assigned a read timestamp of 1, if T writes to i, then i would be assigned a write timestamp of 1;

As outlined by Colouris, Dollimore, & Kindberg, 2001, p. 501:

“A transaction’s request to write an object is valid only if that object was last read and written by earlier transactions. A Transaction’s request to read an object is valid only if that object was last written by an earlier transaction.”

And so the operation conflicts for timestamp order, also outlined by Colouris, Dollimore, & Kindberg, 2001, p. 502, from figure 12.29 are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Rule** | **Tc** | **Ti** | |
| 1 | Write | Read | Tc must not write an object that has been read by any Ti where Ti > Tc. This requires that Tc >= the maximum read timestamp of the object |
| 2 | Write | Read | Tc must not write an object that has been written by any Ti where Ti > Tc. Thiss required that Tc > write timestamp of the committed object |
| 3 | Write | Read | Tc must not read an object that has been written by any Ti where Ti > Tc. This required that Tc > write timestamp of the committed object |

If a read operation of a particular object arrives in a transaction, whereby an earlier transaction has written to that object (tentative version), but the earlier transaction has not yet committed, then the transaction wishing to read the object waits for the earlier transaction to commit before reading. The transaction will then read the object that has been committed .

Tentative values are created when reading an object, along with their read and write time stamps. When a transaction wishes to write an object, new versions of the tentative values are created and once the transaction has committed the actual objects are updated.

In the example on the previous page, when transaction V wishes to read object i, it must first wait for transaction U to finish and commit, before doing this transaction V can’t even prepare to commit. It does not however stop all the other transactions that are going on so transaction W can continue to proceed. In the table on the previous page I’ve shown that transaction W completes all its operations before transaction V restarts, however the actual order of operations doesn’t necessarily have to be this way, transaction T can be restarted as soon as transaction U has committed. For example, transaction U may commit after the read of object m by transaction W, in which V can then be restarted here.

**Part C:**

All ‘write’ operations are stored as tentative versions of objects and other transactions are not able to see these tentative values until a ‘closeTransaction’ is issued, in which the transaction is committed. Each object has its own read and write timestamp and a set of tentative values. A transaction’s read operation is directed to the version with the maximum write timestamp. When a transaction is committed the values of the tentative versions become the values of that object and the time stamps of the tentative versions become the timestamp of the corresponding objects .

A log file is used to store the tentative values of the objects. When a client first requests a read operation of an object, a tentative value of that object, along with its read and write time stamps is created and stored in the log file which is read to determine the order of the transaction and to allow validation rules to be upheld, as well as storing new values to commit once the transaction has finished. If a transaction created before the transaction that read the object tries to write to that object for instance, validation would fail, as the time stamp of any object cannot be less than its previous time stamp and thus that transaction would abort. The timestamp ordering rule is refined to ensure that each transaction accesses a consistent set of versions of the objects and that each transaction commits its tentative versions in order of its timestamps. As the client commits the values on its ‘closeTransaction’ operation, it need not wait when there’s a write request. However the client must wait when read operations need to wait for earlier transactions to finish. This cannot lead to deadlock, since transactions only wait for earlier ones .

When a server receives a request to commit a transaction, it will always be able to do so because all the operations of the transactions are checked for consistency with those of earlier transaction before being carried out, as the committed versions of each object is created in timestamp order, a server sometimes needs to wait for earlier transactions to complete before writing all the committed versions of the objects accessed by a certain transaction. The tentative values are also permanently stored before acknowledging the clients request to commit the transaction, so even if there is a server crash in between where the transactions commit or after all transactions have been committed, the transactions can be recovered .

This is assuming that a transaction is structured such that its writes are all performed at the end of its processing and that all writes of a transaction form an atomic action, where no transaction may execute while a transaction is being written. If this wasn’t the case, then what is called a ‘cascading rollback’ (a chain of rollbacks) can occur. I.e. If Ti aborts but Tc has read a data item written by Ti, then Tc must also abort as if Tc had been allowed to commit earlier, the schedule is not recoverable, further, any transaction that has read a data item written by Tc must also abort. (*Timestamp based concurrency control*. (n.d.). Retrieved March 19, 2008, from cs.ust.hk: [www.cs.ust.hk/~dimitris/CS530/L24.ppt](http://www.cs.ust.hk/~dimitris/CS530/L24.ppt))

The timestamp ordering algorithm is strict; it ensures strict executions of transactions. The read rule will delay a transactions read operation until all transactions that have previously written that object have committed or aborted. The arrangement to commit versions in order ensures that executions of a transaction’s write operation on any object is delayed until all transactions that have previously written that object have committed or aborted .

The entries stored in the log file may look something like this:

|  |
| --- |
| * T = openTransaction, timestamp of T = 1; * T = attempt to read object i; * T = proceed to read object i   + (Tentative value i = object i (value = 10), version = 1, RTS = 1, WTS = 0); * T = attempt to read object k; * T = proceed to read object k   + (Tentative value k = object k (value = 30), version = 1, RTS = 1, WTS= 0); * U = openTransaction, timestamp of U = 2; * U = attempt to read object j; * U = proceed to read object j   + (Tentative value j = object j (value = 20), version =1, RTS = 2, WTS = 0); * V = openTransaction, timestamp = 3; * V = attempt to read object k; * V = proceed to read object k   + (Tentative value k = object k (value = 30), version = 1, RTS = 3, WTS = 0); * U = attempt to write value 44 to tentative value i; * U = proceed to write value 44 to tentative value i   + (Tentative value i = new value (value = 44), version = 2, RTS = 1, WTS = 2); * T = attempt to write value 33 to tentative value l; * T = proceed to write value 33 to tentative value l   + (Tentative value l = new value (value = 33), version = 1, RTS = 0, WTS = 1); * T = prepare to commit; * T = commit; * V = attempt to read object i; * V = denied read of object I;   + V= wait for transaction U to commit; * W = openTransaction, timestamp = 4; * W = attempt to read object m; * W = proceed to read object m   + (Tentative value m = object m, version = 1, RTS=4, WTS = 0); * W = attempt to write value 55 to tentative value m; * W = proceed to write value 55 to tentative value m   + (Tentative value m = new value (value = 55), version = 2, RTS = 4, WTS = 4) * U = prepare to commit * U = commit; * V = restart; * V = attempt to read object i; * V = proceed to read object i   + (Tentative value i = object i (value = 44), version = 1, RTS = 3; WTS = 2); * V = attempt to write value 66 to tentative value k; * V = proceed to write value 66 to tentative value k   + (Tentative value k = new value (value = 66), version = 2, RTS = 3; WTS = 3); * V = prepare to commit; * V = commit; |
|  |
|  |

**Part D:**

In optimistic concurrency control, there is a working phase, in which each transaction has a tentative version of each of the objects that it updates. This is a copy of the most recently committed version of the object, of which is stored in log file, in which the use of these tentative versions allow transactions to abort without any effect to the actual objects. Writes create new tentative values, which are also stored in the log file. When there are several concurrent transactions, several different tentative values of the same object may coexist. Two records are kept of the objects accessed with a transaction, which are a read set and a write set. There is also a ‘validation’ phase of a transaction which uses read-write conflict rules to make sure the scheduling of particular transaction is serially equivalent to all other overlapping transactions which is why it’s important that commit entries in the log file should be ordered by transaction number, which it is assigned when a transaction entered the validation phase (when the client issues a ‘closeTransaction’ request). If the transaction validated finishes, it retains this number, however if it fails validation and is aborted the number is released for reassignment. The transaction number defines a transactions sequence in time which must be in an ascending order in order to achieve serial equivalence. A transaction always finishes its working phase after all transactions with lower numbers. If a later transaction finishes before an earlier one, it must wait for the earlier one to finish before it can be validated .

The log file may look like this:

* Tentative variable x = object I (value = 10);
* Tentative variable k = object k (value = 30);
* Prepare to write value 33 to object l;
* Validation of T successful;
  + Commit, T (write value 33 to object l);
* Tentative variable w = object k 30);
* Tentative variable w = object i (value = 10) ;
* Prepare to write value 66 to object k;
* Validation of V successful;
  + Commit, V (write value 66 to object k)
* Tentative variable z = object j (value = 20);
* Prepare to write value 44 to object i;
* Validation of U successful;
  + Commit, U (write value 44 to object i);
* Tentative variable z = object m (value = 50);
* Prepare to write value 55 to object m;
* Validation of W successful;
  + Commit, W (write value 55 to object m);

# References

Colouris, G., Dollimore, J., & Kindberg, T. (2001). Distributed Systems Concepts and Design.

*Timestamp based concurrency control*. (n.d.). Retrieved March 19, 2008, from cs.ust.hk: www.cs.ust.hk/~dimitris/CS530/L24.ppt