CCTS Project Final Report

Arjun V Anand, CS13B1041 and Akilesh B, CS13B1042

Forward and Backward Optimistic Concurrency Control (FBOCC) Protocol

Introduction:

FBOCC(forward and backward optimistic concurrency control) is a variant of the OCC algorithm. It is basically designed for mobile transactions in wireless broadcast environments.

There are two validation stages.

1. At Clients, a partial backward validation. This is done locally at the clients between between the write set of committed transactions at the server and the read set of running transactions at the client at the beginning of every data cycle. This is done for both read only and update transactions. Any conflicted transaction will be aborted. Successful read only transactions commit locally and update transactions are sent to the server for further processing.
2. At Server, forward validation is performed between the write set of validating transactions and the read set of running transactions. This is done both to transactions generated at the server as well as to update transactions sent to the server by mobile clients. Conflicted server transactions will be aborted. Update transactions has to further undergo final validation at the server in before the forward validation. Conflicted update transactions will be restarted at the client.

FBOCC is designed to minimize the use of the uplink channel in two ways: validation of read-only transactions locally at clients (these constitute the majority of mobile transactions); and early validating and aborting update transactions locally at clients, which makes update transactions more likely to pass the validation and write phases at the server.

Implementation details:

There are basically three type of transactions in the protocol.

1. Mobile Clients : The mobile clients are of two types.
    - Read Only Transactions
    - Update Transactions
2. Server Transactions.

So, I created three different kind of threads (one corresponding to each) and corresponding locks in order to evaluate the FBOCC protocol.

The application which I have created for evaluation is similar to the one given in the past two assignments. I have made some changes so that it suits this client server architecture as required by the protocol. In the application given in the assignment, we don't know the read and write set before. But for partial validation at the Client, we need to know them beforehand.
So, I changed the application Read Only and Update transactions. But the application for Server thread didn't need much change.

The shared array is of size m which all the different transaction tries to read and write.

I created a boolean array broadcast of size m+1 which denotes the broadcast information being sent to the clients. Size, m+1 because there are m data elements and first element will be the id of the server transaction that committed in that cycle. So, as soon a cycle is over (a server transaction commits) the broadcast array is updated.

Backward Validation at Mobile Clients:

At mobile clients, both read only and update transactions have to undergo a partial validation. The algorithm given in the paper is as follows:

Let T_{pv} be the partial validating mobile transaction, and $CD(C_i)$ be the set of data objects that have been committed (updated) in the last broadcast cycle C_i at the server. Let $CRS(T_{pv})$ denote the current read set of transaction T_{pv}, which is the set of data objects that have been read by T_{pv} from previous broadcast cycles. The backward validation is described by the following procedure:

```
partial_validate(T_pv)
{
  if CD(C_i) ∩ CRS(T_pv) ≠ { } then
    abort(T_pv);
  else
  {
    record the value of C_i;
    T_pv is allowed to continue;
  }
  endif;
}
```

Note that $CD(C_i)$ is stored in the control information table. The value of C_i is recorded for the final validation to be described in the next section.

Rot_lock and ut_locks are the locks corresponding to a read only transaction and an update transaction respectively.
I implemented the partialValidate in both rot_lock and ut_lock as follows:

```
      int partialValidate(int id){
            int status=1;
            for(int i=0;i<m;i++){

if(broadcast[i+1]==ut_readSet[id][i]&&broadcast[i+1]==1)
                        status=0;
            }
            if(status==1){
                  ut_commit_cycle[id]=broadcast[0];
            }
            cout<<"\n";
            for(int i=0;i<=m;i++){
                  cout<<broadcast[i];
            }
            cout<<"\n";
           return status;
      }
```

Forward Validation at Server:

At the server, validation of a transaction is done against currently running transactions. Note that a validating transaction at the server may be a server transaction or a mobile transaction submitted by the mobile clients.

For the server transaction it is just a forward validation as done in FOCC. So, I won't go into detail regarding that. For an update transaction submitted by a mobile client, it has to perform a final validation before the forward validation. The final validation is necessary because there may be transactions committed since the last partial validation performed at the mobile client. So, the broadcast cycle number of the last partial validation performed at the mobile client has to be sent to the server along with the update transaction for final validation for which I maintain an array ut_commit_cycle which has the cycle number at which the update transaction was validated partially at the mobile client.

The algorithm given in the paper is as follows:

```
validate(T_v)
{
  if T_v is a mobile transaction then
  {
    final_validate(T_v);
    if return fail then
    {
     abort(T_v);
     exit;
    }
  }
  foreach T_a (a = 1, 2, ..., n)
  {
    if RS(T_a) ∩ WS(T_v) ≠ { } then
     abort(T_a);
  }
  commit WS(T_v) to database;
  CD(C_i) = CD(C_i) ∪ WS(T_v);
}

final_validate(T_v)
{
  foreach T_c (c = 1, 2, ..., m)
  {
    if WS(T_c) ∩ RS(T_v) ≠ { } then
     return fail;
  }
```

I have implemented the above validate in the tryCommit section of the ut_lock as follows:

```
int tryCommit(int id){

            int status=1;
            int intial_commit=ut_commit_cycle[id];
            int min=intial_commit+100;

            for(int i=0;i<history_server.size();i++){
```

```
                if(history_server[i][0]<min)
                        min=history_server[i][0];
        }

        if(min>intial_commit){
                return 0;
        }
        else{
                for(int i=0;i<history_server.size();i++){
                        for(int j=1;j<=m;j++){

if(ut_readSet[id][j-1]==history_server[i][j]&&history_server[i][j]==1
&&history_server[i][0]>=min){
                                                                        return 0;
                                }
                        }
                }
        }

        if(status==1){
                for(int i=0;i<nst;i++){
                        if(st_endTime[i]==0&&st_startTime[i]!=0){
                                for(int k=0;k<m;k++){

if(ut_writeSet[id][k]==st_readSet[i][k]&&st_readSet[i][k]==1)
                                                st_restart[i]=1;
                                }
                        }

                }


        }

        if(status=1){
                //Copying local buffer to shared memory
                for(int k=0;k<m;k++)
                        if(ut_writeSet[id][k]==1)
                                arr[k]=ut_buffer[id][k];


                for(int i=0;i<m;i++){
                        if(ut_writeSet[id][i]==1)
```

```
                        broadcast_buffer[i+1]=1;
                }
        }
        return status;
}
```

Basically, I maintain the write set of all the committed transactions at the server in an array named history_server. First final validation is done with the help of the data in the history_server and then the usual forward validation is done if it passes final validate.

Experimentation:

In the literature, they had compared the performance of the algorithms with other similar algorithms for wireless broadcast environments. But I experimented by increasing the no. of clients in the system keeping the no. of server originated transactions constant. Simply put, I increased the load on the server by increasing the no. of mobile clients which often happens in real life situations.
Parameters taken were as follows:

No. of server threads = 5
Mean delay in each transaction, Lambda = 2 secs
No. of data items =30
consVal=100
I kept no. of read only transactions to be same as no. of update transactions but varied both uniformly from 10 to 50.

Tabulation:

Table 1: Average Commit Delay (in secs) vs No. of threads

Note: Averaged over three runs for each case

No. of Threads	ROT	UT	ST	
10	18.9	123.62	36.4	
20	18.95	128.92	38.2	
30	19.07	133.54	48.4	
40	19.05	138.67	52.2	
50	18.92	142.11		56.4

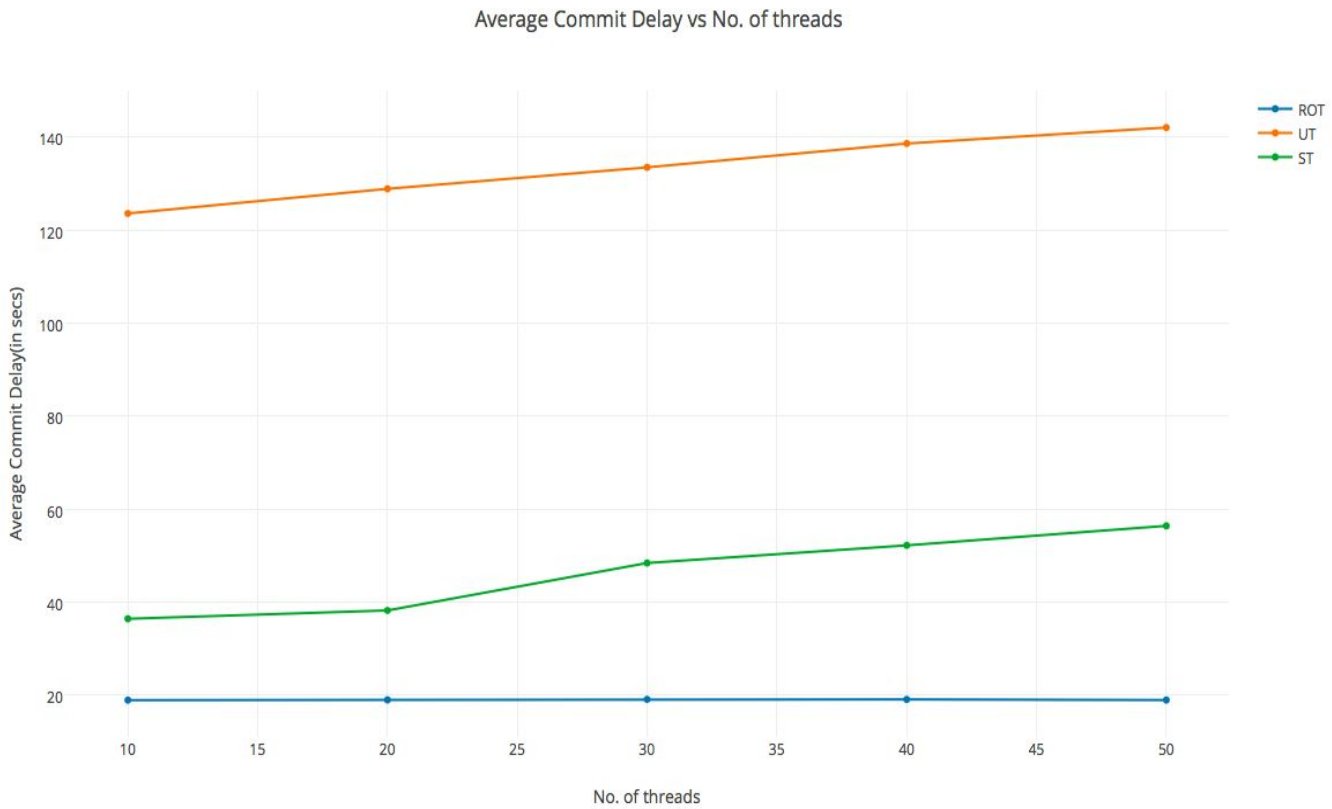
Table 2: Average Abort Count vs No. of threads

Note: Averaged over three runs

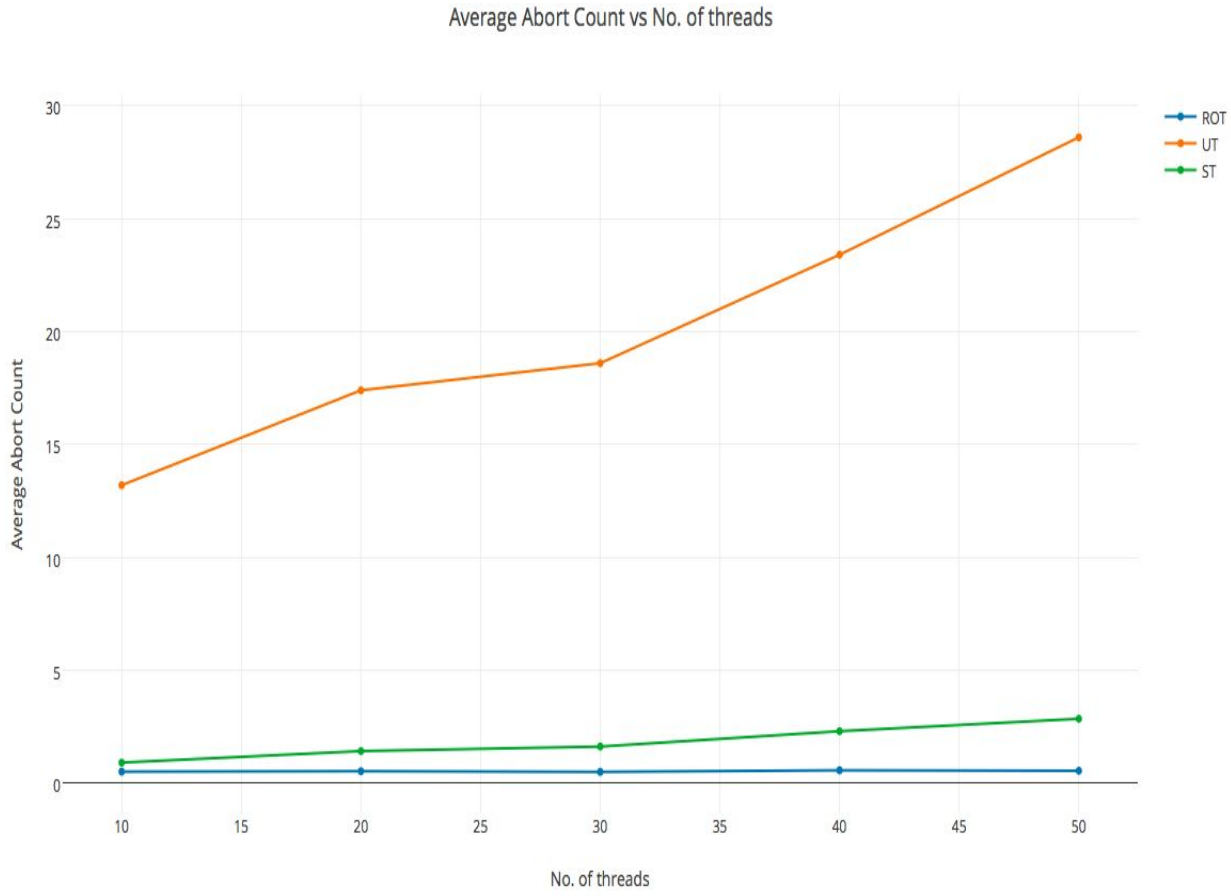
No. of Threads	ROT	UT	ST
10	0.52	13.2	0.92
20	0.54	17.4	1.43
30	0.51	18.6	1.63
40	0.58	23.4	2.31
50	0.56	28.6	2.86

Results:

Graph 1: Average Commit Delay vs No. of threads



Graph 2: Average Abort Count vs No. of threads



Observations:

Both graphs are following the same trend as we have seen in the previous two assignments. Now, lets try to analyse the trend. Let's take the case of average commit delay and the same explanation will hold for average abort count as well.

We can observe that avg commit delay for read only transactions doesn't depend on the no. of threads. This is because increase in no. of clients doesn't affect partial validation because it is done locally at the clients.

We can observe that avg commit delay for update transactions increases with the no. of threads. This is because increase in no. of clients increases the no. of comparisons at the

server during the forward validation because more no. of update transactions pass the partial validation and as a result the chance for a thread getting aborted increases.

We can observe that avg commit delay for server transactions increases with the no. of threads. This is because increase in no. of clients increases the no. of comparisons at the server during the forward validation because more no. of update transactions pass the partial validation and as a result the chance for a thread getting aborted increases.

Update transaction has more average commit delay and average abort count than server transaction because it passes through more no. of validation checks compared to server transactions.

Read only transaction has least average commit delay and average abort count because it passes through only one validation check(partial validation) compared to other transactions which undergo more validation checks.

Read-Write-Validate approach to optimistic concurrency control

An optimistic concurrency control approach, Read-Write-Validate is based on a forward validation approach to address the modern real world demands of databases. The primary goals of this algorithm is to increase energy efficiency and reduce latency. The authors of this paper have showed that this approach decreases the frequency of disk accesses due to contention which in turn leads to enhancement in battery life for the mobile device.

Protocol Description:

Transaction which reaches the end of read phase enters a pre-commit set (PCS). An earliest deadline policy is adopted to chose a member of PCS which enters the write phase. All the transactions which are in their read phase or are members of PCS can be aborted or marked for rerun if they are in conflict with a validating transaction. It is guaranteed that any validating transaction will commit and any other transactions that are in conflict should be rerun.

A forward oriented validation scheme is used which during the validation phase of a transaction i, checks if there is an intersection between the write set (WS) of this transaction (Ti) with the read set of all currently active transactions (Tj). If there is a conflict then:

1) If the conflicting transaction Tj is in its first run, then it is permitted to proceed to its read phase and marked for rerun. Tj is updated with values from other conflicting transactions and then rerun.

2) If Tj is not in its first run then it is aborted. Now, readset of Tj is updated with writeset of Ti. It is rerun again with updated read set.

The correct execution of the arriving transaction is ensured by the following:

1) When a transaction enters the read phase while another transaction is in write phase, there is a chance of reading inconsistent data, which can be detected when the transaction executing in its critical section finishes the write phase and enters the validation phase.
2) Suppose there is a transaction which enters the read phase when another is its validation phase, then any read is made without the updated values (as validation occurs after write). Those transactions which enter the read phase at this point need not be validated against currently validating transaction.

Pseudo Code:

The pseudo-code for the validation phase is as follows:

```
for each Tj in AC do
  if ((WS(Ti) ∩ RS(Tj)) ≠ {}) then
    for each Ok in (WS(Ti) ∩ RS(Tj)) do
      update Ok in CS(Tj);
    end for
    if Tj in initial run then
      mark Tj for rerun;
    else
      update Tj with CS(Tj), rerun Tj;
    end if
  end if
end for
discard WS(Ti);
```

Implementation details

Following additional sets are maintained in comparison to FOCC:

Conflict Set (CS):

CS(Tj) contains the updated read values from validating transactions with which Tj has conflicted. Every item in CS(Tj) is stored until RS(Tj) can be updated. Storing is done instead of updating the read set of Tj straight away, to make sure that writes would not be updated automatically if we decide to update RS(Tj) directly.

Pre-Commit-Set (PCS):

As discussed above, any transaction which reaches the end of read phase enters a pre-commit set (PCS). An earliest deadline policy is adopted to chose a member of PCS which enters the write phase.

Code:

In updtMem, local vectors thr_readSet and thr_readValues are maintained. This is used for keeping track of the order of reads performed by every thread and the corresponding values read.

Only for the first time, randIters is obtained. Subsequent times, just iterate through the size of thr_readSet.
In a similar manner, if its the first run, read function of the rwv class is called passing thread ID and randInd as arguments. If its not the first run, read directly from thr_readValues vector. Conflict set is updated to this value which is read.
Transaction status 0 means aborted, 1 means committed and 2 means active. Lock glock is used whenever global variables are updated. The do while loop in updtMem function is as shown below:

```
do{
           int id=rwv.begin_trans(); // begins a new transaction id
           startTime[ttid]=time(0);
           int randIters= rand()%m;  // gets the number of iterations
to be updated

           // Only for the first run, randIters is random. If not,
just make it equal to the size of thread readset
           if(firstRun != true)
           {
                 randIters = thr_readSet.size();
           }

           //cout << "randIters is " << randIters << endl;

           for(int j=0;j<randIters;j++)
           {
                 //get the next randIndex
                 int randInd=rand()%m;
                 int randVal=rand()%constVal;
```

```
                int locVal;

                if(firstRun != true)
                {
                     randInd = thr_readSet[j];
                }
                else //If it's the first run
                {
                     thr_readSet.push_back(randInd);
                }

                pthread_mutex_lock(&glock);
                int temp1 = statusThread[ttid];
                pthread_mutex_unlock(&glock);

                if(temp1 == 0)
                     break;

                //Read a shared memory
                //If it's the firstRun
                if(firstRun == true)
                {
                     locVal=rwv.read(ttid, randInd);

                     //Since readSet, conflict_set are global, lock
should be used
                     pthread_mutex_lock(&glock);
                     readSet[ttid][randInd]=1;
                     conflict_set[ttid][randInd] = locVal; //update
conflict set to locVal
                     pthread_mutex_unlock(&glock);

                     thr_readValues.push_back(locVal);
                }
                else
                {
                     locVal = thr_readValues[j];

                     //Since readSet, conflict_set are global, lock
should be used
                     pthread_mutex_lock(&glock);
                     readSet[ttid][randInd] = 1;
```

```
                    conflict_set[ttid][randInd] = locVal; //update
conflict set to locVal
                    pthread_mutex_unlock(&glock);
               }


               time_t readTime = time(0);

               //cout << "Before read print \n";
               //Print lock
               pthread_mutex_lock(&lock2);
               cout<<"Thread id "<< ttid <<" Transaction "<< id<< "
reads from " <<randInd<<" a value "<<locVal<<" at time "<< gmtime(&
readTime)->tm_hour<<":"<<gmtime(& readTime)->tm_min<<":"<<gmtime(&
readTime)->tm_sec<<"\n";
               pthread_mutex_unlock(&lock2);
               locVal=locVal+randVal;
               rwv.write(ttid, randInd,locVal);

               pthread_mutex_lock(&glock);
                    temp1 = statusThread[ttid];
               pthread_mutex_unlock(&glock);

               if(temp1 == 0)
                    break;


               // request to write back to the shared memory
               pthread_mutex_lock(&glock);
               writeSet[ttid][randInd]=1;
               pthread_mutex_unlock(&glock);
               time_t writeTime = time(0);
               pthread_mutex_lock(&lock2);
               cout<<"Thread id "<< ttid <<" Transaction "<< id<< "
writes to " <<randInd<<" a value "<<locVal<<" at time "<< gmtime(&
writeTime)->tm_hour<<":"<<gmtime(& writeTime)->tm_min<<":"<<gmtime(&
writeTime)->tm_sec<<"\n";
               pthread_mutex_unlock(&lock2);
               long randTime=round(distribution1(gen));

               // sleep for a random amount of time which simulates
some complex computation
               sleep(randTime);
```

```
            }
            //cout << "after for loop\n";

            pthread_mutex_lock(&glock);
            pre_commit_set[ttid] = 1;
            // int temp = statusThread[ttid];
            pthread_mutex_unlock(&glock);

            pthread_mutex_lock(&lock);
            pthread_mutex_lock(&glock);
                  int temp = statusThread[ttid];
            pthread_mutex_unlock(&glock);

            if(temp == 2)   //If the thread ttid is currently active
            {
                  for(int k=0;k<m;k++)
                        if(writeSet[ttid][k]==1)
                              arr[k]=buffer[ttid][k];

                  status=rwv.tryCommit(ttid); // ttid is thread is, id
is transaction id
                  committed[ttid]=status;
                  statusThread[ttid] = 1;
                  time_t cTime = time(0);
                  pthread_mutex_lock(&lock2);
                  cout<<"Thread id " << ttid << " Transaction "<< id<<
" tryCommits with result "<< status <<" at time "<<gmtime(&
cTime)->tm_hour<<":"<<gmtime(& cTime)->tm_min<<":"<<gmtime(&
cTime)->tm_sec<<"\n";
                  pthread_mutex_unlock(&lock2);

                  endTime[ttid]=time(0);
            }
            else
            {
                  for(int i=0;i<m;i++)
                  {
                        readSet[ttid][i]=0;
                        writeSet[ttid][i]=0;
                        buffer[ttid][i]=0;
                  }
                  statusThread[ttid] = 2;         //Make it active
```

```
                startTime[ttid]=0;              //Reinitialize
startTime
                cout<< "Thread " << ttid << " transaction "<< id<<"
aborted \n";

                for(int k=0; k<thr_readSet.size(); k++)
                     thr_readValues[k] =
conflict_set[ttid][thr_readSet[k]];
           }
           pthread_mutex_unlock(&lock);
           abortCnt++;   // Increment the abort count

           cout << "value of status is " << status << endl;
     }while(status!=1);
```

The code for tryCommit function

```
     long* endTime_temp = new long[n];
     long* startTime_temp = new long[n];

           int writeSet_temp[n][m];
           int readSet_temp[n][m];

           //Creating temporary copies of readSet, writeSet,
endTime and startTime which is useful for checking condition in
validation phase
           //without modifying the actual original values.
           pthread_mutex_lock(&glock);
           for(int j=0; j<n; j++)
           {
                endTime_temp[j] = endTime[j];
                startTime_temp[j] = startTime[j];

                for(int k=0; k<m; k++)
                {
                     writeSet_temp[j][k] = writeSet[j][k];
                     readSet_temp[j][k] = readSet[j][k];
                }
```

```
          }
          pthread_mutex_unlock(&glock);

          //If the transaction being evaluated doesn't satisfy
the conditions of RWV, commit is set to 0.
          for(int j=0;j<n;j++)
          {
               if(endTime_temp[j]==0 && startTime_temp[j]!=0 &&
j!=id)
               {
                         for(int k=0; k<m; k++)
                         {

if(writeSet_temp[id][k]==readSet_temp[j][k] &&
readSet_temp[j][k]==1)
                                   {
          cout << "Thread "<<j<<" is going to be aborted
"<<endl;

pthread_mutex_lock(&glock);
          statusThread[j] = 0;
          conflict_set[j][k] = buffer[id][k];
pthread_mutex_unlock(&glock);
                                   }
                         }
               }
          }
```

In tryCommit, before validating a transaction temporary local copies of readSet, writeSet, endTime and startTime are created because these can be changes made to these sets by other transactions which are currently active but not validating.

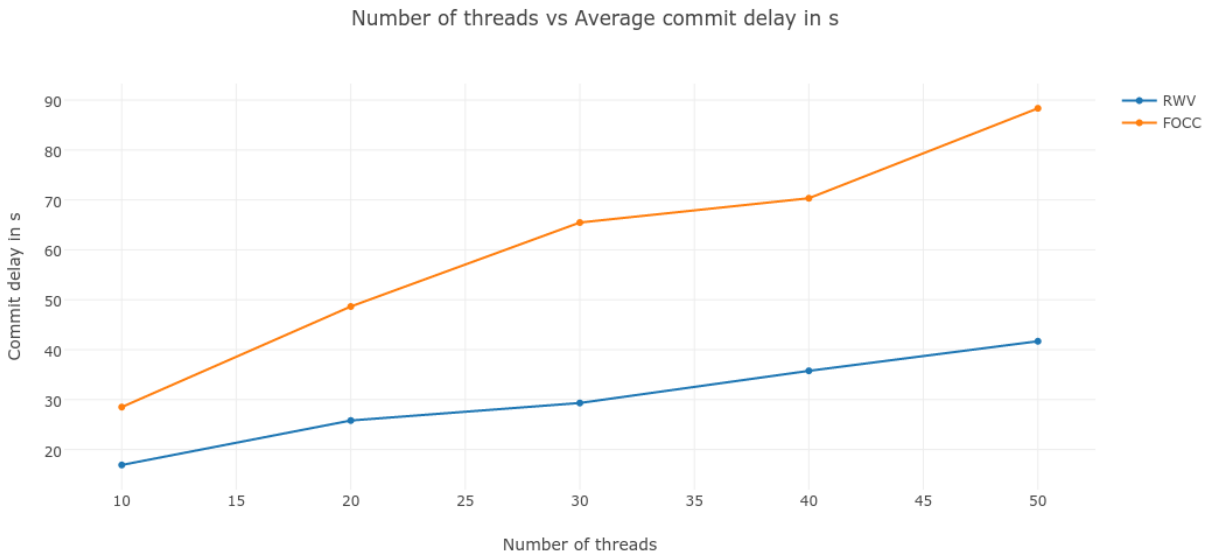
Results:

RWV protocol was compared with FOCC protocol.

Average Commit Delay (in secs) vs No. of threads

Note: Averaged over three runs for each case

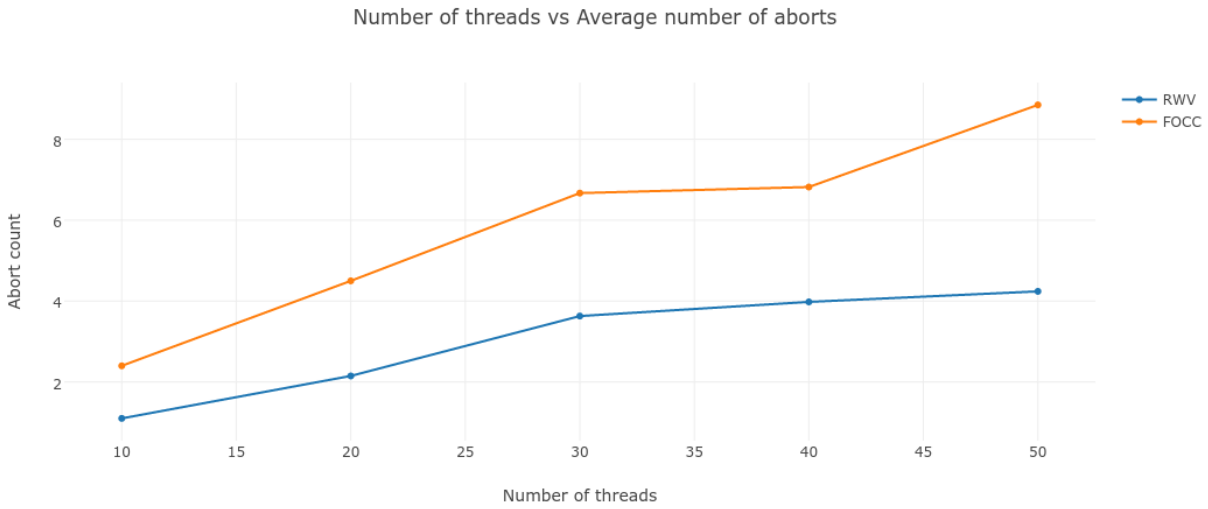
No. of threads	RWV	FOCC
10	16.9	28.5
20	25.81	48.65
30	29.32	65.47
40	35.76	70.34
50	41.69	88.35



Average Abort Count vs No. of threads

Note: Averaged over three runs

No. of threads	RWV	FOCC
10	1.1	2.4
20	2.15	4.5
30	3.63	6.67
40	3.98	6.82
50	4.24	8.85



Conclusion:

RWV is found to have lower commit delay as well as lower number of abort counts than FOCC. That is, RWV protocol is found to perform better than FOCC. Some advantages of moving write phase before the validation phase are:

i) Writes become visible to transactions which are in the read phase before, which leads to more chance of reading latest data.

ii) Blocking may get reduced. In FOCC, transactions which are in the read phase needs to be blocked because a transaction commits a change to the database. This blocking is not necessary in RWV as older and out-of-date reads are detected by the subsequent validation step.