Contents

1 Correctness

1.1 Transactions

We have a set of Threads T each representing a transaction.The code for a Transaction is like below:

```
Thread.doIt(new Callable)(){
// code for transaction
}
```

This is adopted from dstm and can and may be changed. Within this block which represents a member of set T. The functions beginTransction and endTransaction are implicitly called by the system without the intervention of the programmer.However, the programmer can read and write from TransactionalFile Objects within the blocks of a member of T. To provide consisteny the read and write from ordinary Java object files should be prohibited, otherwise the semantics of transactions would not be preserved. Hence, the sequence of operations provided to the programmer within a member of T is the set {TransactionalFile.Read(), TransactionalFile.Write()i, TransactionalFile.Seek()}. However, always a beginTransaction() and commitTransaction() (if the transaction does not abort prior to this point) are performed too.

A Transaction consists of a sequence of operations. Thus, a transaction can be represented as the {op_1, op_2, ..., op_n}. The flow of program should be in a manner that we could have an arbitrary serial order of the members of T (e.g. T_5, T_6, T_1,...) or any other order. The serial order should be in

such a way that for any given two members of the set T, it looks all the operations of one occur before all the operation of the other.

1.2 Defenitions

textbfDef 1- *Set of Primitive Operations*: Operations are taken from the set {forcedreadoffset(filedescriptor), writeoffset(filedescriptor), readdata(inode, offset, length), writedata(inode, offset, length), commit}. We denote read operations as readoffset and readdata and write operations as writeoffset and writedata.

Def 2- *Operations Sharing Resources* 1- A forcedreadoffset operation and writeoffset operations are said to be sharing resources if they both operate on the same filedescriptor. 2- A readdata and writedata operations are said to be sharing resources if they both operate on the same inode AND the range of (offset, offset + length) within one overlaps with that of the other.

Def 3- A read operation can only see the writes made by write operations sharing reources.

Def 4- *Commit Instant and Commit Operation*: A transaction commits when it invokes the "commit" operation. A transaction T_i is said to "commit" at instant t_i, if and only if it reflects its "writes" made by write operations in the filesystem or equivalently makes it writes visible to the whole system for all members of T accesing the data at instant t_j such that $t_j > t_i$. After a transaction invokes commit operation, the transactin ends and no more operation by transaction is done. An operation $\in OP_{T_i}$ is said to commit if and only if T_i commits. Commit instant are not splittable, hence it appears to other transactions that all writes are refleted in the file system together.

Def 5- *Precedence Relationship*: If $OP_{executed} = \{op_0, op_1, ..., op_n\}$ represnts the operations executed so far(from all transactions running), and the indices represnt the order of operations executed so far in an ascending manner, we define $op_i \rightarrow op_j$ (precedes) if and only if $i < j$.

Def 6- *Visbility of Writes*: Assume T_i is an uncommitted transaction and $T_{commited}$ indicates the set of commited transaction (those which have invoked the "commit" operation). A read operation $b \in OP_{T_i}$ MAY ONLY see the writes(changes) made by preceding write operation a that shares resources with b AND also is subject to one of the following conditions:

1- $a \in OP_{T_j}$ $(j \neq i)$ such that $T_j \in T_{commited}$

2- $a \in OP_{T_i}$ and a happens before b in the natural order of the transaction. Formally, $a \rightarrow b$

Corrolary 1- *No See in The Fututre*: If $\exists op_i \in WRITEOP_{T_i}$ and $\exists op_j \in READOPT_j$ such that $i \neq j$ and $op_i \rightarrow op_j$ then if $op_j \rightarrow commit - operation_{T_i}$, writes made by op_j are not seen by op_i.i

proof: Follows immediately from Def 5 and Def 6.

Corrolary 2- *Read Must See Most Recent Comitted Write*: If $a \in OPT_i$ reads the resource r_i at t_i. In case a has multiple writer precedessors sharing r_i (denoted as $OP_{precedessors-for-r_i}$), then if all of them are from sets other than OP_{T_i}, a sees the writes made by $op_{T_j} \in OP_{precedessors-for-r_i}$ such that T_j has the greatest "commit instant" less than t_i between all transaction with such operations . Otherwise, a sees the most recent precedessor in OP_{T_i}.

axiom: Assume a is accessing data at t_i, according to Def 6, only writes by those transactions that have already committed would be visble. Hence, t_i is neccasarily gretaer than all members of $\{committime_{T_1}, ..., commitime_{T_{i-1}}\}$,and hence the writes to r_i made by these commited transaction, have accroding to Def 4 been already reflected in the file system. Since, T_{i-1} has the greatest commit instant its changes have overriden that of previously commited transactions and thus, a reading r_i from file system sees these changes.

In case 2 that there are writer operations writng r_i and preceding a in OP_{T_i}, according to defenition of transaction, r_i should be read from the writes made by write operations in T_i, and as the last of such write operation overrides the written data to r_i by other operations, a gets to see the most recent precedessor in T_i.

Def 7- *Precedence Relationship For Transactions:* $\forall$ $op_{T_i} \in OP_{T_i}$ and $\forall op_{T_j} \in OP_{T_j}$ if and only if $op_{T_i} \rightarrow op_{T_j}$ then $T_i \rightarrow T_j$ (this defines precedes relationship for members of T)

Def 8- *Correctness:* A sequence of transactions are said to be consistent if and only if a total ordeing of them according to precedence relationship can be established that demonstrates the same behavior as the execution of the program. Behavior for an operation means the data it has read or wants to write. Demonstrating the same behavior thus means all the read operations should still see the same data in the new sequence as they have seen in the actual sequence. However writs always writes the same value no

matter what. Hence the behavior of a write operation is not alterable.

Note 1: Def 7 and Def 8 indicate that if operations can be commuted in a given sequnce of $OP_{executed} = \{op_{T_1}(0), op_{T_4}(0), ...op_{T_1}(n)\}$ such that a total ordering of transactions (e.g $\{OP_{T_1}, OP_{T_2}, ..., OP_{T_n})\}$ can be obtained then the execution is consistent and correct. The eligiblity to commute is subject to conforming to Corrolary 1 & 2.

Def 9- *Relocation Operations in A Sequence*: In a sequence of operations $OP = \{op_1, op_2, ..., op_n\}$, any operation can relocate its position in the sequence unless as a result of this change, the behavior of a read operation changes (it reads different data).

Note 2: *No Commute for Operations Belonging to The Same Transaction*: If $op_i \in OP_{T_i}$ and $op_{i+1} \in OP_{T_i}$, we never commute op_i and op_{i+1}, since even if this exchange does not change any operations behavior, there is still no point in doing this as the aim is to put all operations belonging to the same transaction together, the internal order among these is not any of our concern, and the precedence relationship among these as indicated by the execution should be maintained.

Note 3: *Precedence Relationship Between Commit Operations Should Be Preserved*: As a rquirement we want to have the notion that transaction are executed in the serial order imposed by their commit operations. Hence, commit operations can not be commuted.

All the rules explained later are based on the assumptions made in Note 2 & 3, hence these two types of commution are ruled out by default.

1.3 Rules

Rule 1- $\forall op_i \in OPT_i$, $\forall op_j \in OPT_j$, $\forall op_k \in OPT_k$, if $op_i \rightarrow op_k$ and $op_k \rightarrow op_j$, then $op_i \rightarrow op_j$

proof: Follows from the defenition of $\rightarrow$

Rule 2- If $T_i \rightarrow T_k$ and $T_k \rightarrow T_j$ then $T_i \rightarrow T_j$

proof: Follws from Rule 1 and Def 7.

Rule 3- *Exchnaging Position of Consecutive Operations*: Formally having the sequence of operations $OP = \{op_1, ..., op_i, op_{i+1}, ..., op_n\}$, if $op_i \in OP_{T_n}$ and $op_{i+1} \in OPT_m$ such that $n \neq m$, op_i and op_{i+1} can exchange

positions if and only if none of the follwing conditions apply:

1- If $op_i = commit$ and $op_{i+1} = read$ and op_{i+1} reads r_k, and $\nexists op_k$ in OP_{T_m} such that writes to r_k, and if there $existsop_l \in OP_{T_n}$ such that writes the data r_k, then op_i and op_{i+1} can not exchange positions.

2- If $op_{i+1} = commit$ and $op_i = read$ and op_i reads r_k and $\nexists op_k$ in OP_{T_n} such that writes to r_k, and if there $existsop_l \in OP_{T_m}$ such that writes the data r_k, op_i and op_{i+1} can not exchange positions.

Proof: According to Def 9, none of the operatios in the sequence should change behavior as the result of this exchange, however in this argument only op_i and op_{i+1} may change behavior as those are the only operations that their postions in the sequence is changed. However, since behavior is only defined for read operations, one of these without losing generality lets say op_i should be a read operation on agiven resource r_k.

Now changing the (op_i, op_{i+1}) to (op_{i+1}, op_i) changes the behavior of op_i if and only if op_{i+1} writes r_k at the file system (all the previous precedessors are still the same, only $opi + 1$ has been added). This means by defenition the op_{i+1} should be a commit operation. And, if op_{i+1} is a commit operation for T_m, and there is at least one write operation in T_m writing to r_k, then according to Corrolary op_i should see the most recent results and hence if there is no precedessor for op_i in T_n itself that writes to r_k, then op_i sees the changes made by the write operarion in T_m. These changes could not have been seen in the first case (op_i, op_{i+1}) due to Corrolary 1 (No See in The Future).

Rule 4 -*Relocating the Position of an Operation Within the Sequence*: Given a sequence of operations $OP = \{op_1, ..., op_i, op_{i+1}, ..., op_j, op_{j+1}, ..., op_n\}$, op_j can be put into the standing $i < j$ within the sequence (resulting in $OP = \{op_1, ..., op_i, op_j, op_{i+1}, ..., op_{j+1}, ..., op_n\}$) if and only if $\forall op \in \{op_{i+1}, ..., op_{j-1}\}$, op and op_j belong to different transactions and the pair (op_j, op) or (op, op_j) is not a pair subject to one of the conditions in Rule 3. The same holds true for $i > j$.

proof: We use induction to prove if assumptions above hold true op_j can be relocated to $i = j - n$. Assume the same OP as before. If $n = 1$ then since according to assumption the pair (op_j, op_{j-1}) is not subject to the conditions in Rule 3, these two can be easily exchanged. Now, lets assume the op_j can be relocated to $j-n-1$, now we prove it for n. After relocation to $j - n - 1$, op_{i+1} immediately precedes op_j, as according to assumption and Rule 3 op_j and op_{i+1} can exchange positions. After this exchange, op_j has been relocated by n and to i and op_i now immediately precedes op_j.

Now we prove the other side of the argument, that if op_j can be relocated in i, $\forall op \in \{op_{i+1}, ..., opj-1\}, (op_j, op)$ is not a pair subject to the condition in Rule 3. Lets assume there is op in T_i such that (op, op_j) is a $(readr_n, commit)$ and the commit invlolves making a write in T_j to r_n durable and a precedessor writer that writes to r_n does not exists for op in OP_{T_i} , now if op_j is relocated to position i, op_j would precede op and hence op would see the writes by the operation in T_j (defenition of commit Def 4 & Corrolary 2) and hence the behavior of op would change as it does not read the same data as before (the data read before could not have been the same thing due to Corrolary 1). If the pair of (op, op_j) is $(commit, readr_n)$ the same reasoning would do.

The whole argument can be used to prove the Rule for $i > j$ as well.

Rule 5-*Operation in the Set of Exeeceuted Operations Belonging to Committed Transactions Should Be Able To Precede Those in The Transaction About to Commit*: $OP_{executed} = \{op_1, ..., op_n\}$ represents the set of executed operations before instant t_j and $T_{committed}$ represents the set of committed transactions committed successfully before t_j. If T_j invokes *commit* operation at instant t_j - then T_j commits at instant t_j if and only if operations in OP can be commuted in a way such that $\forall op_i \in OP_{Tcommitted}$, $\forall op_j \in OP_{T_j}$, $op_i \rightarrow op_j$.

proof: If all those operations can be commuted tp precede those in T_j we could have $OP_{executed} = \{op_1, ..., OPT_j\}$. This by defenition of transaction means T_j can commit (it is executed in its whole entierty).

Now we have to prove T_j commits only if $\forall T_i \in T_{committed} T_i \rightarrow T_j$. Now we'll show that if all committed operations can not precede operation of T_j the T_j can not commit. $OP = \{op_{T_j}(1), ..., opT_j(m)\}$ represent the operations in T_j excluding the *commit* in the order they have occured in $OP_{executed}$. Assume $\{op_{T_j}(k), ...opT_j(m)\}$ can be relocated in $OP_{executed}$ in the standing {n-(m-k), ...,n-1) but $op_{T_j}(k-1)$ can not. This means first of all $op_{T_j}(k-1)$ is a *read* operation reading r_n(each transaction has only one *commit* operation).

Furthermore, this implies there is a *commit* operation by some transactione T_i between $opT_j(k-1)$ and commit operation by T_j and $\exists opinOPT_i$ such that writes r_n and $\nexists op_i \in OPT_j$ such that $op_i \rightarrow op_{T_i}(k-1)$. On the other hand since there is a *commit* by T_i in the middle, the commit by T_J can not be commuted in a way so OP precedes it without any operation belonging to other transaction in the middle (two *commits* can not commute). Hence we can not have a sequence where all operations belonging to T_j are

located next to each other, this contradicts the defenition of transaction, hence the transaction can not commit.

2 An Abstract Model For The System

2.1 Assumption For The System

Def 10- *Set of User-Level Operations*: User-Level Operations are taken from the set {Read(fildescriptor), GetFilePointer(filedescriptor), Write(filedescriptor), Seek(filedescriptor), EndTransaction}.

Note 4- *Assignment Operations Need Not Be Shown In* $OP_{excecuted}$: Operations like offset = offset + length and other assignment operations in OP_{T_i} need not be shown in the actual sequence of operations namely $OP_{executed}$ that consists of operations executed by different transaction so far, the reason is simply all such operations are local to the transaction and do not affect any other transaction's state and hence do not restrict the commution of other operations in any manner.

Note 5- *Forced-Readoffset(fd)*: Reads the offset for the filedescriptor and makes the transaction bound to this value.

2.2 Transaction Dependency on FileDescriptor Offset

Each filedescriptor has an associated offset with it, within each transaction this offset can be in 4 different states, these states indicate the dependency the transaction has on the value of this offset:

1- **No Access**: This is the default state for all filedescriptors in a transaction and is changed as soon as there is an access to the the descriptor within the transaction (any of the use-level operations are invoked).

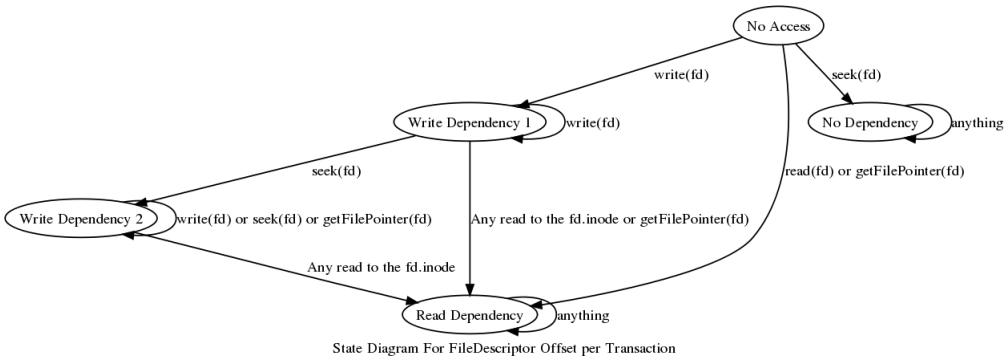
2- **None**: Meaning there is no dependency for any operation in this transaction on the value of the offset associated with this descriptor regarding other transactions.

3- **Write Dependency**: This kind of dependency means there is at least one operation in OP_{T_i} having an unknown offset(essentially a write operation) value as argument. The value of this unknown offset will be determined at commit instant.

4- **Read Dependency**: This kind of dependency means there is at least one operation acting on an offset value for fd, where the value for fd has

been determined by a previously committed transaction.

The state machine below depicts the behavior of the user level operations regarding how the offset corresponding to that transaction changes:



State Diagram For FileDescriptor Offset per Transaction

Explanation: Whenever the offset status for the transaction goes to "Read Dependence", a "forced-readoffset(fd)" operation is issued immediately preceding the operation that caused this transformation. The forced-radoofset(fd) is only issued if there is not a forced-readoffset(fd) in the OP_{T_i} already.

Axiom For Diagram: If the first access to a filedescriptor is a $Seek(fd)$, then the following operation on fd gets the offset value from the assignment made by the $Seek(fd)$ and advances the offset. Hence, the following operations get this offset and the offset value the filedescriptor had had before this transaction accesses fd is never referenced (thats why it is an absorbing state). This conforms to the definition of "None" state.

If $Read(fd.inode, offset, length)$ or $GetFilePointer(fd)$ is the first access made by this transaction, the offset is read (since the data needs to be read at this instant), this offset should be the one committed by a previously committed transaction (as this is the first access to fd in this transaction). Once the offset is read, it is always dependent on this value (hence an absorbing state). For all the following operations, the offset value is known. This conforms to the definition that there is at least (the first read ever on fd by this transaction) one operation that acts on the offset value for fd and rules out the "Write Dependency" and "None" states.

If $Write(fd.inode, offset, length)$ is the first access made to fd by this transaction, then the offset to write to, can be decided at commit instant since the Write functions means start writing at the most recent committed fd.offset, hence offset realization can be postponed till commit instant. Any

Writes or Seeks would still leave this dependency, since operations after a Seek act on absolute offset, and Writes preceding any Seek can all determine the offset at commit instant for the same reason as before.

However, if a Read on the same fd in the transaction is invoked there are two possibilities:

1- A Seek precedes this Read, hence the offset value is absolute and is not read, however the ranges that are supposed to be written by Writes preceding the Seek, may overlap with the range Read is willing to Read from, and according to Rules(Most Recent Changes Should Be Visible) if thats the case the Read should be able to see this data, this suggests the ranges that all the Writes are going to write to should be realized now and this requires settling down on a value for all file descriptors offsets for this inode at this instant. Based on these, the most recent committed offset value for all these descriptors should be assigned to the offsets for the Writes that for the first time accessed the descriptor. Other for writes preceding the Seek, get this value as being advance by prior writes.

2- No Seek precedes the Read, hence the offset value the Read has to read from is unknown, since preceding Writes to fd have all used unknown offsets, the offset value given to Read is an unknown once, however it has to be known, follows that the offset value for the Write that for the first time accessed this fd should be decided upon and as shown before, the value should be the most recent committed offset value for fd. The offset value for this read or other writes, is the offset value obtained as being advanced by those operations.

The two same possibilities exist when a GetFilePointer operation is invoked on the same fd:

1- If a Seek precedes it, then the offset value becomes absolute and hence the getFilePointer could retrieve the value assigned by Seek.

2- Otherwise, the offset value is still unknown, hence to be able to determine the offset value at this instant, the value obtained by reading the last committed offset value should be assigned to the offset value. for the first Write to fd.

2.3 User-Level Operations Structure

The user-level operation cab be broken as follows:

1- Seek(fd): This operation sets the offset for filedescriptor. We define it as demonstarted below:

Just an internal assignment in the transaction, $\{fd.offset = value\}$.

2- Write(fd):

1- $\{writedata(fd.inode, offset, length), fd.offset = fd.offset + length\}$

3- GetFilePointer(fd):

$\{\{forcedreadoffset(fd)$ issued as demonstrated at the state diagram if any, it is issued when the state for fd in this transaction is not in "No Dependency" or "Write Dependency 2"$\}\}$

4- Read(fd):

$\forall filedescriptor fd_i$ where $fd_i.inode = fd.inode$ and the state for fd_i in this transaction is not $NoDependency$, Read(fd) $=\{\{forcedreadoffset(fd_i)\}\}, readdata(fd.in$
$offset + length\}$

5- EndTransaction:

$\forall fd_i$ such that the state for fd_i in this transaction is not "No State", $EndTransaction = \{\{writeoffset(fd_i)\}, commit\}$

Essentially for any fd that the correspondent state diagram in the transactions shows is in a state other than "No Access", a writeoffset(fd) is issued while committing.

2.4 *writedata* Dependecy on Offset

Any writedata operation within a transaction gets fd and an offset as arguments. There writes are reflected in the commit instant, however the offset to write to as we saw earlier for some writes i s determined at commit instant and for other is bound to a specific value before commit instant. We should have a policy to be able to diffrenciate between these two. The 3 rules below odes this.

1- If the state for fd a given distinguish in a transaction is "Write Dependency 1" all writes by that transaction to that fd will get the value of the offset to write to at commit instant (since in "Write Dependency 1" all

writes are at unknown offsets) .

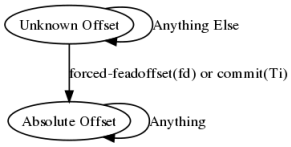
2- If the state for fd is "Read Dependency" or "No Dependency" then all writes on fd within this transaction should be done at offset determined for them when they were invoked (since all writes are to known offsets either determined by the transaction itself or a previously committed transaction).

3- Otherwise, if the state is "Write Dependency 2", then if the $writedata(fd.inode, ...)$ operation precedes a $Seek(fd)$ then the offset is determined at commit instant (since all such writes are at unknown offset). Otherwise, the writes should be done at the offset decided upon earlier (the write after a Seek write the offset determined by Seek and hence to a known offset).

We could also think of this as a state machine for each writedata operation. The state diagram is created for each operation when it is first invoked and is subject to two things:

1- If there $\exists Seek(fd) or forced-readoffset(fd) \in OP_{T_i}$ such that those precede the $writedata(fd.inode, ...)$ then the initial state in the state diagram for this writedata is Absolute.

2- Otherwise the initial state is Unknown offset.



State diagram for FileDescriptor Offset per writedata(fd.inode, offset, lenght) operation in Ti

The final state for all $writedata$ operations is Absolute, since the write should be perfomed at a specefic offset eventually. However, depending on the prevuious circumstances the system would immediately prior to commit determine the offset or would have realized it earlier.

2.5 Guidelines for Implementaion

As we saw earlier in Rule 4, any two operations can commute across each other unless they are subject to one of the two conditions in Rule 3.

Guideline 1: *Commuting forced-readoffset Operations*: A $forced - readoffset(fd) \in OP_{T_i}$ can go past a $commit_{T_j}$ if and only $\nexists writeoffset(fd) in OP_{T_j}$.

Axiom: It follows immediately from Rule 4 and conditions in Rule

3, that this $forced - readoffset(fd)$ can go pats the commit. What remains to be proven is $/\exists writeoffset(fd) \in OP_{T_i}$ such that it precedes $forced - readoffset(fd)$, as this would mean even if $\exists writeoffset \in OP_{T_j}$ still the $forced - readoffset(fd)$ could commute with $commit_{T_j}$.

This stems from the definition of EndTransaction operation, and the state diagram. A $forced - readoffset$ can be issued at any place in OP_Ti however it would always precede the $writeoffset(fd) \in OP_{T_i}$ since this is last operation in OP_{T_i} before *commit*.

Guideline 2: *Commuting readdata Operations*: A $readdata(fd_i.inode, offset_i, lenght_i) \in OP_{T_i}$ can go past a $commit_{T_j}$ if and only if

1- $\nexists writedata(fd_j.inode, offset_j, length_j) in OPT_j$ such that $fd_j.inode = fd_i.inode$ and the two ranges $(offset_i, offset_i + length_i)$ and $(offset_i, offset_j + length_j)$ have an intersection.

OR

2- $\exists$ a set of $writedata(fd_j.inode, offset_j, lenght_j) in OP_{T_i}$ such that for all of them $writedata \rightarrow readdata$ and $fd_i.inode = fd_j.inode$ and the range $(offset_i, offset_i + length)$ is a subrange for a combiantion of ranges for these *writedata* operations.

Axiom: This follows imeediately from Rule 4 (1 and 2 are concrete representations for conditions 1 and 2 in Rule 3). However, it should be noted that whenever a *readata* operation wants to commute across a *commit* operation $\in T_j$, if $\exists writedata \in OP_{T_j}$, the offset for the *wtitedata* operation is known at this instant (according to the state diagram demonstarted below and regardless of whether $i = j$ or not), hence it would be trivial to see whether these *writedata* operations intersect (or include in case they are in the same transaction as the *readata*) with the *readdata* or not.

Guideline 3: *Reads Should be Validated At Commit Instant*: If an operation a that "reads" some data r_n (reads the data at some $t < t_{commit-of-the-transaction}$(the commit instant)), it should be ensured that the data r_n is still valid in the actual file system at $t_{commit-of-the-transaction}$ (commit instant) or T_i has to abort.

Axiom: If the data is not valid anymore it means the data r_n has been written since t_{i-1}. This implies at least one operation b has written the data since the use and b does not precede a and a precedes b. b is either an operation in the same transaction or a different one. However, if b is in the same transaction, then the data read is valid at commit instant by

defenition. On the other hand, assume b belongs to a different transaction namely T_j. Since T_j should have already commited, according to Corrolary 2 all operation in OP_{T_j} should precede those in OP_{T_i}, however, we know there is at least an operation b that does not precede a (since a has not seen the "writes" made to r_n by b). Hence, T_i can not commit.

Guideline 4: *If Reads Are Valid At Commit Instant, the Transaction Commits*: If by applying Guideline 3 for T_i it is ensured that $\forall r_i \in RT_i$ (all data read by T_i) is still valid at commit instant, then $\forall T_j \in T_{committed} T_j \rightarrow T_j$ and hence according to Rule 4 T_i commits.

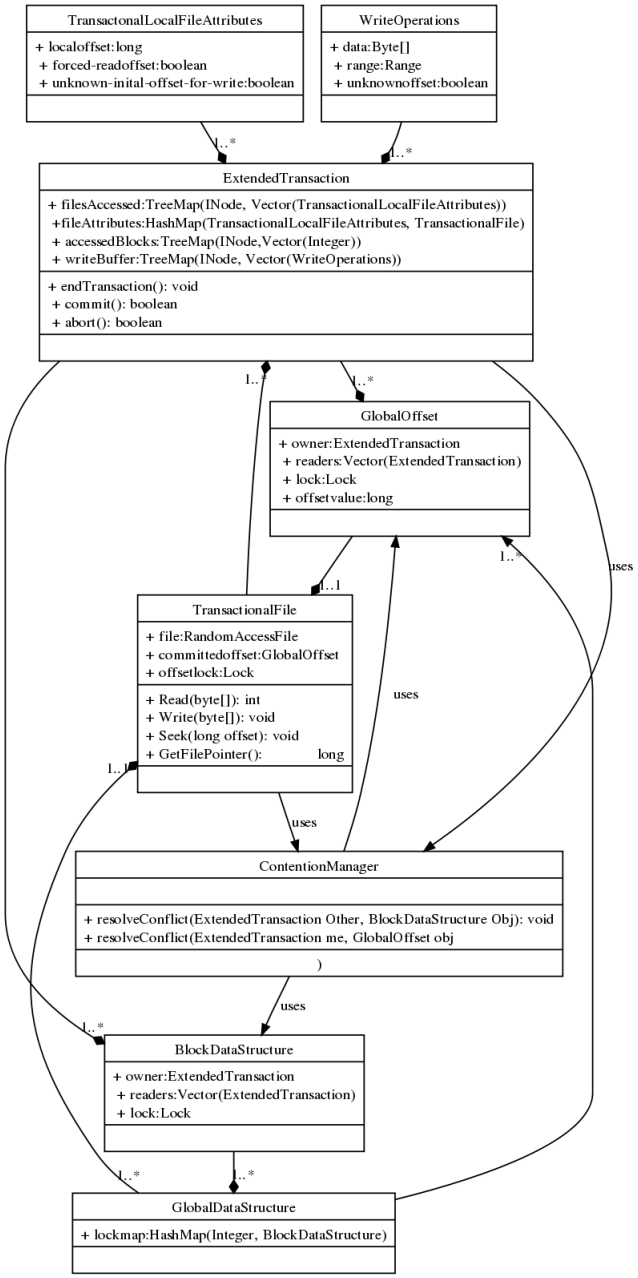
Axiom: If all data read is still valid at commit instant, means all operation in the set of operations belonging to committed transactions, can be relocated to precede those in T_i (since no writes have been seen), and consequently all those transactions precede T_i. Rule 5 ensures such transaction would be able to commit.

Guideline 5: *A Transaction About to Commit Can Abort Other Transaction Safely*: A transaction about to commit can check to what resources it has alreay written (denote this as R_i), an then $\forall T_j in T$ such that T_j is active at this instant can check to see if any of those have read $r_i \in R_i$, if so and no other operation in OP_{T_j} preceding this read has written to r_i then T_i can abort T_j, since T_j is doomed to abrot anyway (according to Guideline 3).

3 System Implementation Overview

3.1 Components

In this sextion we will show how our system conforms to the guidlines and rules required by the system. Only the main components of the system are depicted in the figure and listed below. It should be noted that even for theses classes the major functions and fields are exposed in the illustration.



TransactionalFile: This is the object most user-level operations are perfomed on. It can be either created inside or outside a transaction. Accessing a TransactionalFile by T_i means T_i has invoked a user-level operation on the object. A Transactionalfile can be shared among any subset of the members of T. If shared, the offset is shared between these as well.

ExtendedTransaction: The class and data structures required to do

the actions required by a transaction (commit, abort, ...).

TransactionalLocalFileAttributes: The class to maintain the data structures specific to a transaction regarding an already accessed TransactionalFile object (offset status for the transaction, inital offset status for each Write operation and etc.).

GlobalOffset: The class to maintain the global offset assosiated with each TransactionalFile object (the value, who owns the offset, who reads the offset, etc.)

GlobalDataStructure: The class to maintain the global data structure assosiated with each inode (such as mapping of block locks for each inode).

ConflictManager: The manager to resolve the conflict, this can be when a transaction can not successfully commit or a read or write has to do subsidary actions regarding other transaction or conflict over locks and etc.

BlockDataStructure: The data structure representing a block in an inode. GlobalDataStructure uses instances of this class to preserve the info rmation.

WriteOperation: The structure of a writedata operatin, including the range it is supposed to write, the data to be written and wether this is an absolute or unknown offset value.

3.2 Algorithm

In this section we examine the psudecode for user-level operations in the propsed system and show how our system conforms to the formalisatiom buitl up in earlier sections.

tf_i**.Read(data[])**

If (tfla = currenttransaction.accessedfile.contains(tf_i))
for any of the writes in writeuffer to tf_i.inode
if [tfla.localoffset, tfla.offset + data.length] overlaps with range for the write
copy the intersect portion to the correspondent portion of data[]
if any non-filled portion of data[] exists
try to lock tf_i.offset.lock
try to lock all the blocks within range [tfla.localoffset, tfla.offset + data.length]

if all locks succeed then read the non-filled portions from tf_i.file
if not succeed consult contentionManager to resolve the conflict
else add tf_i.inode to files accessed by this transaction and create
the corresponding TransactionalLocalFileAttributes and call Read(data[])