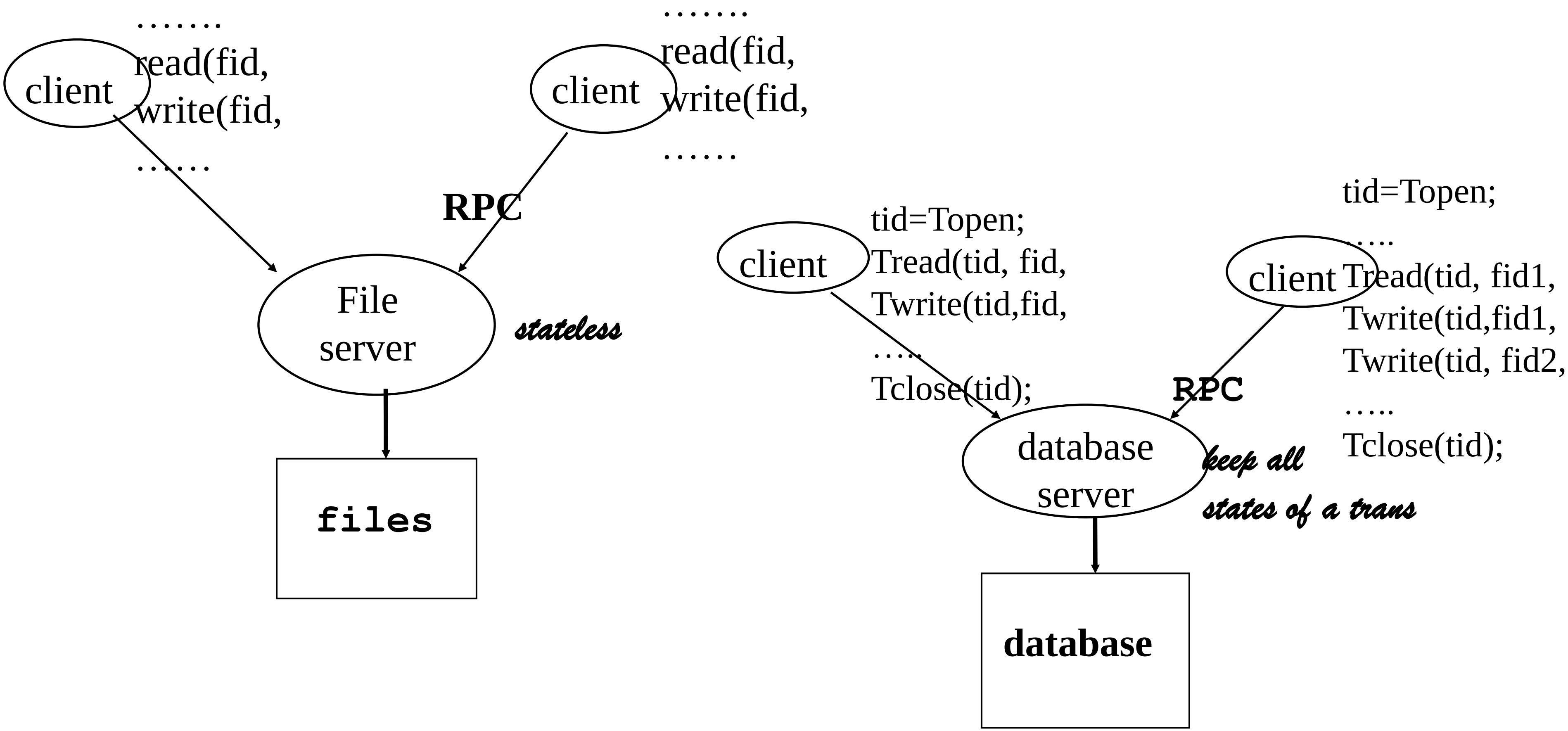
Distributed Systems
分布式系统

Transaction Processing Systems

事务处理系统

File operations and Transaction operations



Definitions of Atomic Transactions

Atomic Transaction: a sequence of data access operations that are atomic in the sense of:

1. All-or-Nothing
2. Serializability

The **ACID** properties of a transaction in RM-ODP:

- Atomicity
- Consistency
- Isolation
- Durability

What makes Database inconsistent?

- Concurrent accesses of transactions
- Failures of servers

Two well-known problems caused by concurrency:

Lost update problem

Transaction T: transfer(A, B, 4);		**Transaction U:** transfer(C, B, 3);	
Balance := A.Read()	$100		
A.Write (balance-4)	$96		
		Balance := C.Read()	$300
		C.Write(balance-3)	$297
Balance := B.Read()	$200		
		Balance := B.Read()	$200
		B.Write(balance + 3)	$203
B.Write (balance +4)	$204		

Inconsistent Retrieval Problem

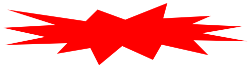
Transaction T: transfer(A, B, 100);		**Transaction U:** TotalBalance()	
Balance := A.Read()	\$200		
A.Write (balance-100)	\$100		
		Balance := A.Read()	\$100
		Balance:= balance+B.Read()	\$300
		Balance:= balance+C.Read()	\$300+
Balance := B.Read()	\$200	.	
B.Write (balance +100)	\$300	.	

Problem Caused by Server Failures

Transaction: transfer(A, B, 100)

bal = A.Read();

A.Write(bal - 100);

 ← server crash point

bal = B.Read();

B.Write(bal + 100);

Transactional File Operations

Transactions provide a programming environment:

- Concurrency transparency
- Failure transparency

Transactional file service operations:

TID = OpenTrans ()

(Commit, Abort) = CloseTrans (TID)

AbortTrans (TID)

TWrite(TID, FID, i, Data)

TRead(TID, FID, i, buf)

FID = TCreate(TID, filename, type)

TDelete(TID, FID)

……

An Example of Using Transactions

Transaction: transfer (A, B, 100)

```
tid = OpenTrans();
        Tread(tid, A, bal)
        Twrite(tid, accountA, bal-100)
        Tread(tid, accountB, bal)
        Twrite(tid, accountB, bal+100)
CloseTrans(tid);
```

How to achieve atomicity of transactions?

Failure Recovery (guarantee nothing-or-all)

- Intention list approach
- Shadow version approach

Concurrency Control (guarantee serializability)

- 2-Phase Locking
- Timestamp Ordering
- Optimistic Method

Failure Recovery of Transactions

An execution of a transaction has Two Phases:

1. **Tentative phase**, the execution transaction body
2. **Commit phase**, making tentative values permanent

Making transactions failure recoverable:

- Keep the tentative values of data on disk that can survive failures
- Restore data items at the restart from a failure (recovery operation should be idempotent)
- Make the commit phase repeatable

Intention List Approach (failure recovery)

Example of intention list:

tid = OpenTrans;

TWrite(tid, fd1, len1, data1);

TWrite(tid, fd2, len2, data2);

Close(tid)

Intention List:

tid, status,

{"Twrite, fd, pos1, len1, data1",

"Twrite, fd, pos2, len2, data2"}

Implementation of Intention List

Transaction operations by using intention list:

- *Twrite* writes data to the intention list.
- *Tread* reads data from the intention list if it is present.
- *CloseTrans* performs operations in the intention list onto database files (or the recovery file).

Recovery manager: a program (part of the server) which is called when the server restarts from a failure.

Recovery file: a file used by the recovery manager to restore the database to a consistency state. Each entry of the file, for one transaction, contains the information:

- Tid
- transaction status
- intention list

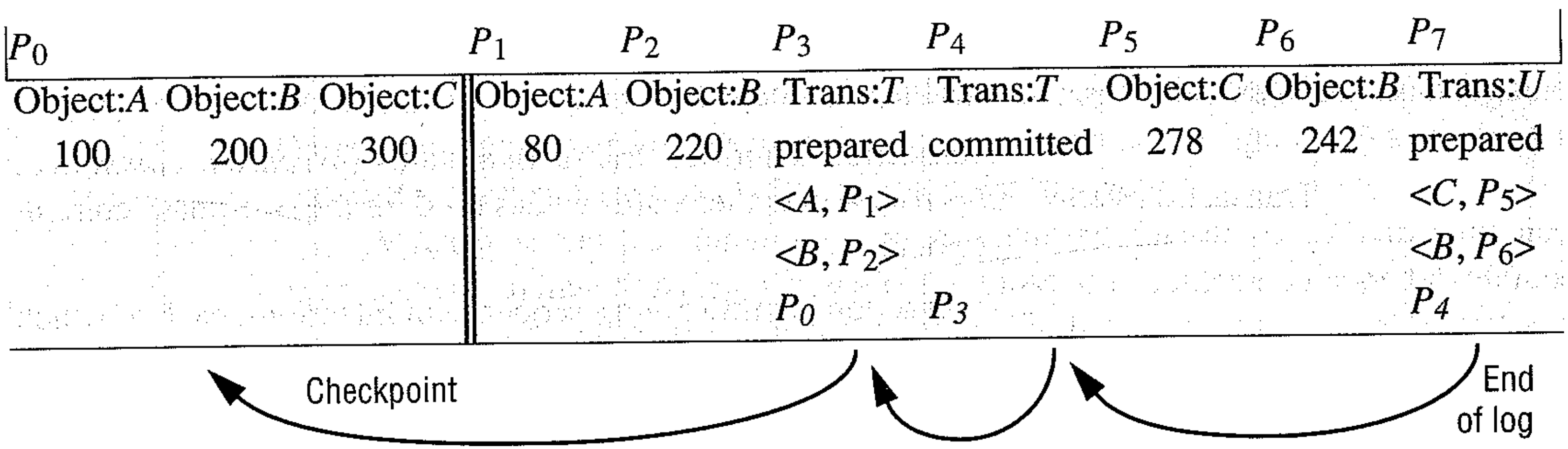
Example of recovery

Transactions T and U:

T: transfer (*A*, *B*, \$20)

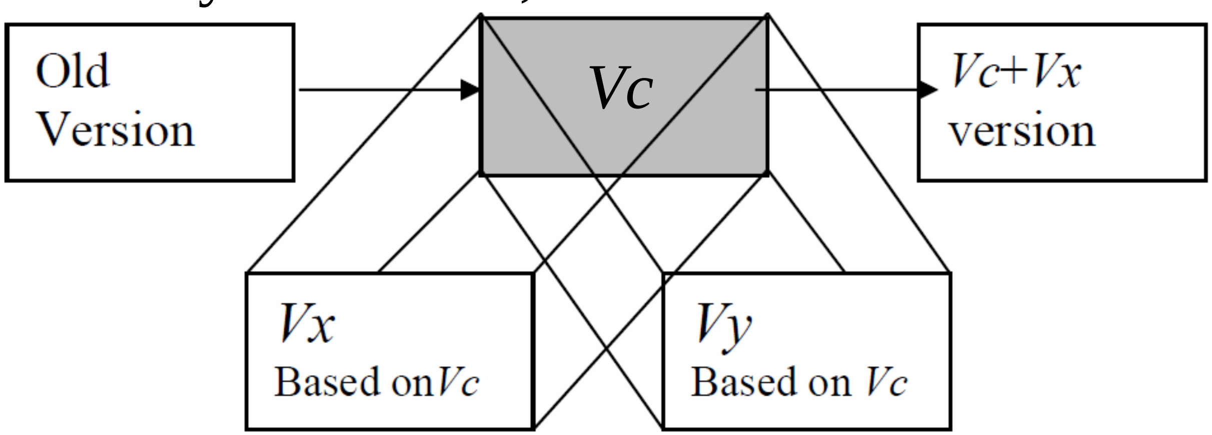
U: transfer (*C*, *B*, \$22)

Recovery: remove tentative data created by U (p_5 & p_6) and commit the data updated by T (p_1 & p_2) to database.



Shadow Version Approach (failure recovery)

- When a transaction modifies a file or data item, it creates a shadow (tentative) version of the file.
- The subsequent *Twrite*/*Tread* are performed on the shadow version.
- At *closeTrans*, it detects version conflict with other concurrent transactions (done by concurrency control).
- If no conflict, the shadow version is merged with other concurrent versions already committed; otherwise it is aborted.



Implementation of Shadow Version Approach

- A copy of the file index block is created at the first *Twrite* of a transaction.
- Each *Twrite* operation creates shadow pages. The index entries of modified pages are made pointing to the shadow pages.
- The original file index block is replaced by the new one at the commit of the transaction (it’s an idempotent operation).

Serializability (Serially Equivalent)

- Two transactions may conflict with each other when they access the same data item(s).
- Only write operations may cause inconsistency.

Def. Serializability: $T \rightarrow U$ (T completes before U starts):

1. read operations in U must read the data written by T
2. write operations in U must overwrite the data done by T
3. T cannot see any effect of U

Absolute Sequential Execution

T → U :

T	U
read(a)	
write(a)	
read(b)	
write(b)	
	read(c)
	read(b)
	write(b)

U → T:

T	U
	read(c)
	read(b)
	write(b)
read(a)	
write(a)	
read(b)	
write(b)	

Serial Equivalence

$T \rightarrow U$:

T	U
	read(c)
read(a)	
write(a)	
read(b)	
write(b)	
	read(b)
	write(b)

$U \rightarrow T$:

T	U
read(a)	
write(a)	
read(c)	
	read(b)
	write(b)
read(b)	
write(b)	

Non-serial Equivalence

T	U	
read(a)		
write(a)		
read(b)		
	read(c)	
	read(b)	
	write(b)	T → U
write(b)		U → T

Non-serial Equivalence of Lost Update and Inconsistent Retrievals

	Transaction T: transfer(A, B, 4)		Transaction U: transfer(C, B, 3)		
	Banlance := A.Read()	$100			
	A.Write (balance-4)	$96			
			Balance := C.Read()	$300	
			C.Write(balance-3)	$297	
	Balance := B.Read()	$200			
			Balance := B.Read()	$200	
			B.Write(balance + 3)	$203	T→U
U→T	B.Write (balance +4)	$204			

	Transaction T: transfer(A, B, 100);		Transaction U: TotalBalance()		
	Banlance := A.Read()	$200			
	A.Write (balance-100)	$100			
			Balance := A.Read()	$100	T→U
			Balance:= balance+B.Read()	$300	
			Balance:= balance+C.Read()	$300+	
	Balance := B.Read()	$200	.		
U→T	B.Write (balance +100)	$300	.		

Serializability of Transactions

Conflicting operations: Two operations access the same data item and one of them is *write*.

Conflicting transactions: Two transactions that have conflicting operations in them.

Serializability of transactions T_1 and T_2:

$T_1 \rightarrow T_2$ iff $\forall$two conflicting operations $op_{1i} \in T_1$ and $op_{2j} \in T_2$: $op_{1i} \rightarrow op_{2j}$.

Serializability of Transactions:

The execution of a set of transactions T_1 T_2, ..., T_m, is equivalent to the execution of them in a serial order, i.e., $T_{i1} \rightarrow T_{i2} \rightarrow$, ..., $\rightarrow T_{im}$

Note: Serializability is the minimum requirement for database consistency in general.

2-Phase Locking

First phase: obtaining locks.

Second phase: releasing locks.

When a transaction begins to release a lock, it cannot apply for locks any more.

Why two phases of locking?

Server operations for T	Server operations for U
write(a);	
read(b);	
	read(b);
	write(b);
write(b);	

Simple locking won't work (2-Phase Locking)

First phase: obtaining locks.

Second phase: releasing locks.

When a transaction begins to release a lock, it cannot apply for locks any more.

Think about by using simple locking:

Server operations for T	Server operations for U
lock(a);write(a); unlock(a)	
lock(b); read(b); unlock(b)	
	lock(b); read(b); unlock(b)
	lock(b);write(b); unlock(b)
lock(b);write(b); unlock(b)	

2-Phase Locking

First phase: obtaining locks.

Second phase: releasing locks.

When a transaction begins to release a lock, it cannot apply for locks any more.

Use two phases of locking:

Server operations for T	Server operations for U
lock(a);write(a);	
lock(b); read(b);	
	lock(b) – wait!!! read(b);
	……. write(b);
lock(b);write(b); unlock(a, b)	

Note: U cannot obtain lock on *b* until T completes.

Serializability of 2-Phase Locking

Serializability: All transactions are serialized in the order of the time they obtain locks on data items.

Lock compatibility:

For one data item		*Lock requested* *Read*	*Write*
Lock already set	*None*	OK	OK
	Read	OK	Wait
	Write	Wait	Wait

Implementation of Locking

Lock manager: a module of a server program. It maintains a table of locks for the data items of the server.

Each entry in the table of locks has:

- transaction ID
- data-item ID
- lock type
- a condition variable (a queue for clients waiting unlock)

Lock manager provides two operations:

- lock(trans, dataItem, lockType)
- unlock(trans), signal the condition variable

The *lock* and *unlock* operations must be atomic

Deadlock in 2-Phase Locking

Deadlock may occur in 2-Phase locking

Deadlock prevention and handling:

1. Lock all data items a transaction accesses at the start of the transaction.
2. Set timeout for waiting a lock.

Discussions on 2-Phase Locking

Advantages of locking:

1. No abort and restart.
2. Simple for understanding and implementation.

Disadvantages:

1. Pessimistic.
2. Poor efficiency (cost of locking and clients waiting for locks).
3. Deadlock.

Timestamp Ordering

Serializability of Timestamp Ordering: transactions are serialized in the order of their start time (*openTrans*).

Important Timestamps:

1. Each transaction is assigned an unique timestamp *T* (also used as *TID*) at the open.
2. Each data item has a write-timestamp (*wt*) and a read-timestamp *(rt*).

Timestamp Ordering (Cont'd)

Read/Write Operations: the transaction's timestamp T is compared with the *rt* and *wt* of the data to decide if the operation can proceed.

<u>Read Rule</u>:

- transaction T can read a data item only if the data item was last written by an earlier transaction.
- if a *read* of T is accepted, T becomes **a tentative** *rt* of the data (if T > *rt*).

<u>Write Rule</u>:

- transaction T can write the data only if the data was last read and written by earlier transactions.
- when a *write* of T is accepted, a tentative version of data is the created with timestamp T;

<u>Commitment</u>:

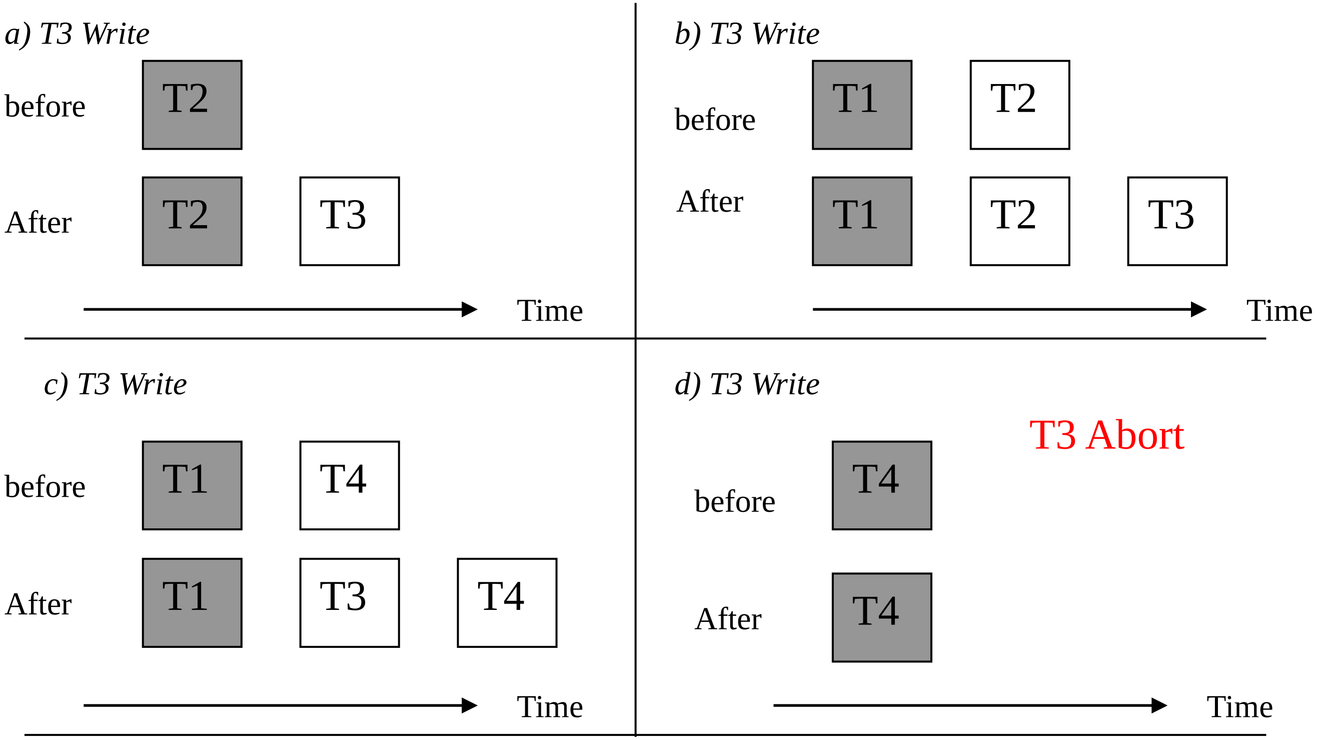
- when T commits, its tentative version of data and tentative *rt* become permanent;
- if T aborts, all tentative data and *rt*s created by T are removed.

Rules for Write

W1: T ≥ *rt* and T > *wt*, a tentative value is created.

W2: T < *rt* (including tentative *rt*) or T < *wt*, abort.

T1 < T2 < T3 < T4

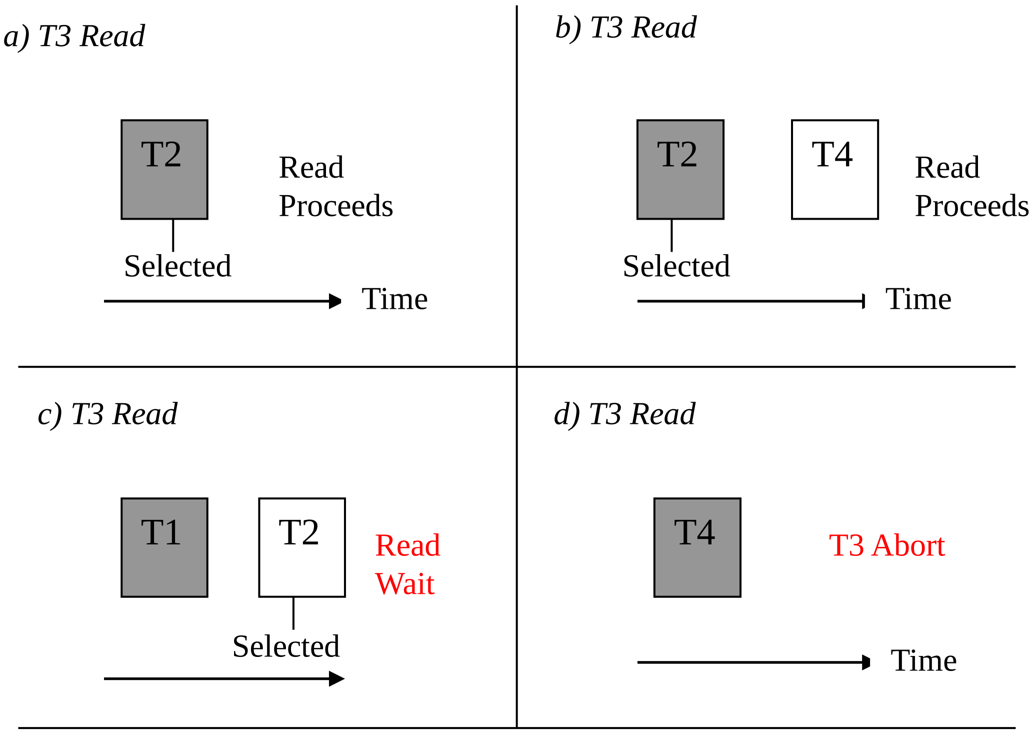


Rules for Read

R1: T ≥ *wt*

- if there is a tentative value made by itself, read this tentative value; otherwise:
- if there are tentative values whose timestamps are earlier than T, wait for the tentative values committed; otherwise:
- read data immediately.

R2: T < *wt*, abort.



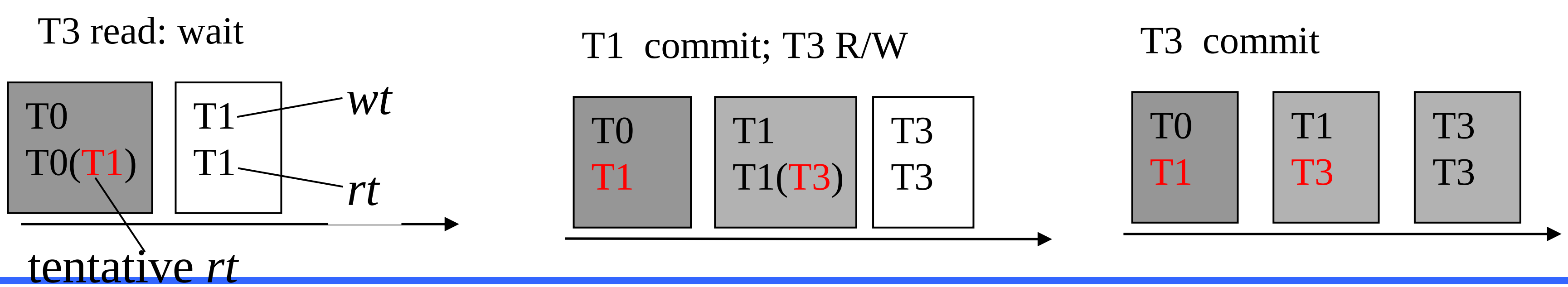
Commitment of Transactions

- The commit of a tentative value (including a tentative *rt* of a data object) has to wait for the commit of tentative values of earlier transactions.
- A transaction waits for the earlier transactions only, which avoids deadlocks.

Note: all transactions are serialized in the order of the time when they start (*openTrans*).

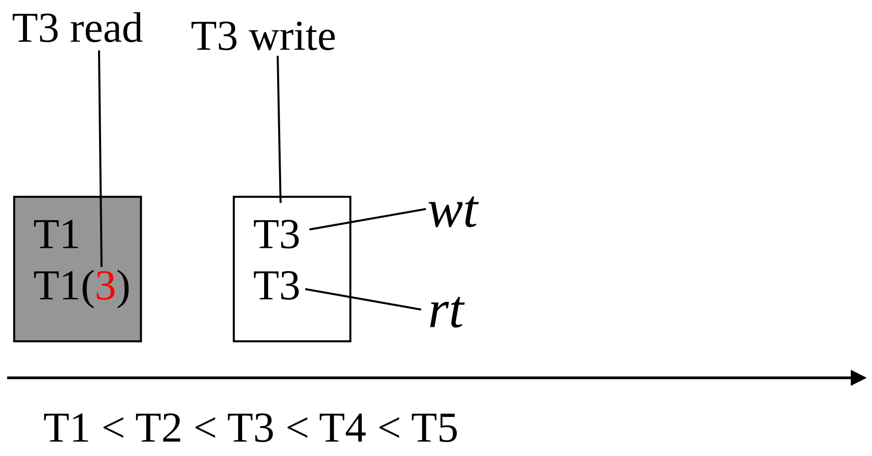
Multiversion Timestamp Ordering

- The server keeps old committed versions (a history of versions) as well as tentative versions in a list of versions of the data.
- A *read* that arrives too late need not be rejected. It returns the data whose version has the largest *wt* that is less than the transaction. A read still need to wait for a tentative version to commit/abort.
- There is no conflict between *write* operations, bcs each transaction writes on its own version of the data. But a write will be rejected if the data was read by a later transaction.
- A commit does not need to wait for earlier transactions (bcs multi-versions can co-exist) if it does not read data from earlier versions.



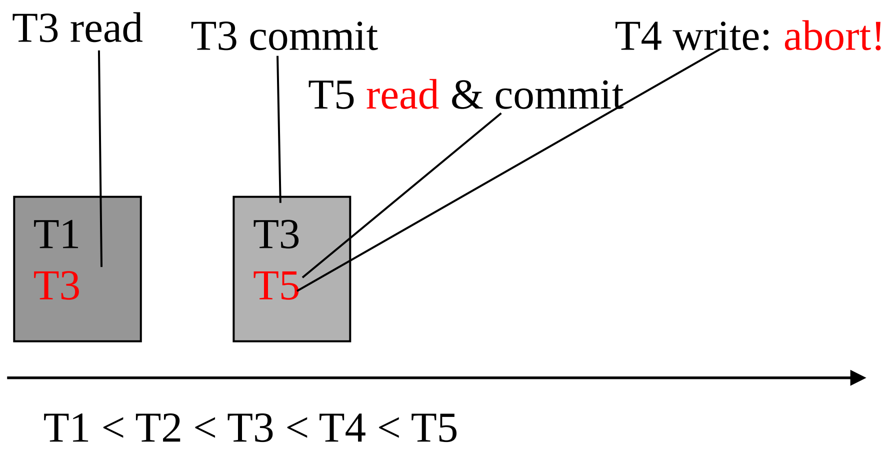
Multiversion Timestamp Ordering: late read / write

- In multiversion timestamp ordering, a read operation can always proceed (provided the old version is still kept)
- But a write operation arrived too late can still be rejected.
- Example: after T3 read & write…. (cont'd)..



Multiversion Timestamp Ordering: late read/write (cont'd)

- Ater T5 reads the data and committed, if T4-write arrives, it'll be aborted; otherwise you have conflict of T4 & T5.
- Rule: a write operation arrived too late can still be rejected.



Discussions on Timestamp Ordering

Advantages:

- No deadlocks.

Disadvantage:

- Extra storage for timestamps.
- Possible abort and restart.
- A long transaction may block others.

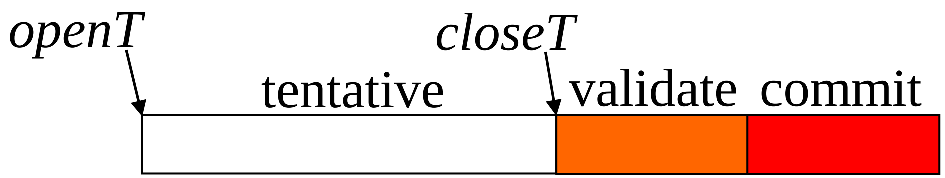
Timestamp ordering method is pessmistic (over-strong):

T	U
t_1=openTrans	
	t_2 = openTrans
	read(t_2, b)
	write(t_2 ,b)
	closeTrans(t_2)
read(t_1, b)	← T will **abort** (unnecessarily)

Optimistic Method (Optimistic Timestamp Ordering)

Observation: the possibility of conflicts of two transactions is low. Transactions are allowed to progress as if there were no conflict at all.

A transaction has three phases: tentative, validation and commit.



Tentative phase: read and write are returned immediately (write is to a tentative version). Read set (RS) and Write set (WS) of the trans are recorded.

Validation phase: detect the conflicts with other concurrent transactions and decide commit/abort as the result.

Commit phase: commit tentative values made by the transaction.

Note: transactions are serialized in the order of their close time.

Validation Condition

When validating T_i, system checks each T_j which is concurrent with T_i (i.e. T_i starts in between the start and completion of T_j).

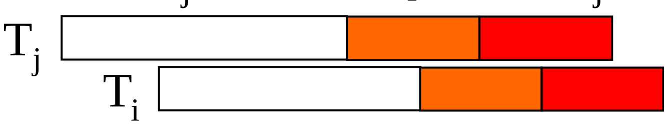
1. Sequential validation (overlap in tentative phase):

$WS(T_j) \cap RS(T_i) = \varnothing$

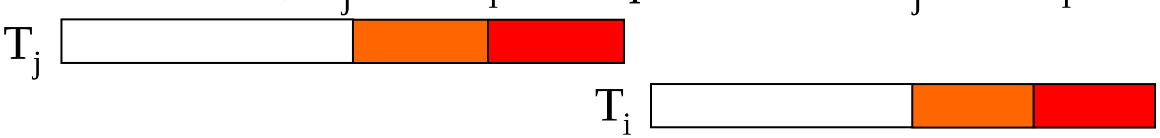
2. Concurrent validation (overlap in validation/commit phase):

$WS(T_j) \cap (RS(T_i) \cup WS(T_i)) = \varnothing$.

(if $WS(T_j) \cap WS(T_i) \neq \varnothing$, T_j may overwrite T_i's data due to concurrent commit)



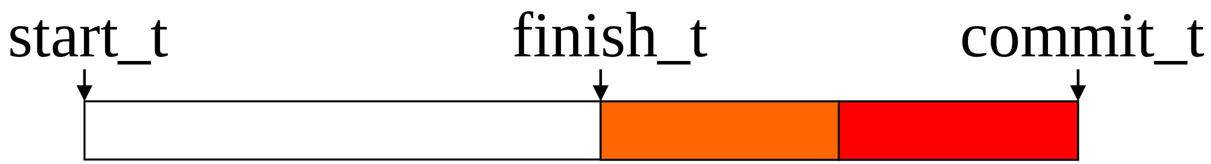
In either case, T_j and T_i are equivalent to $T_j \rightarrow T_i$.



Implementation Details

Three important time points of a transaction:

start_t, finish_t and commit_t.

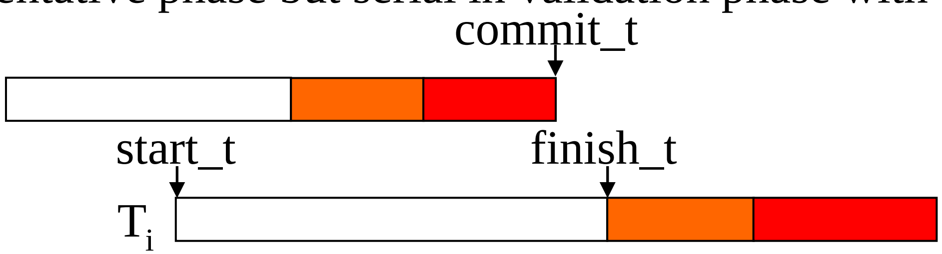


validation_set: a set of transactions being in validation.

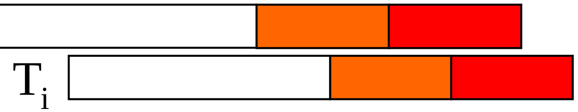
Note: start_t is the time of "openTrans", finish_t is the time of "closeTrans".

Validation, Write, and Commit of Transaction T_i

1. When a T_i enters validation, record time finish_t, make a copy of validation_set and add T_i into validation_set.
   - transactions whose commit_t is in between start_t and finish_t of T_i are concurrent in tentative phase but serial in validation phase with T_i.



   - transactions in validation_set are concurrent in validation phase with T_i.

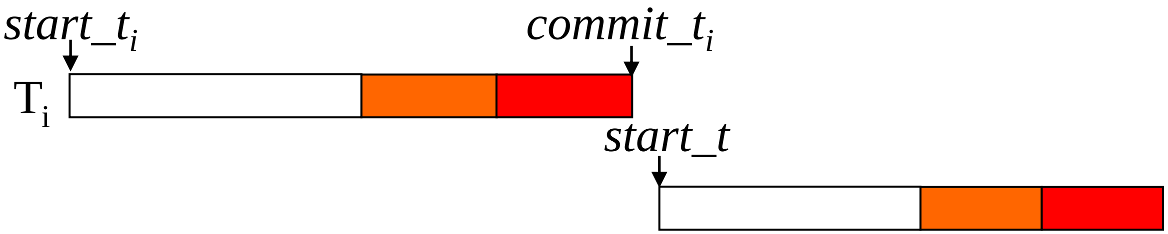


2. If T_i passes validation, it enters write-phase.
3. T_i's commit_t is recorded and T_i is removed from validation_set.

How long we need to keep information of T_i

Information of committed transactions should be kept for other transactions validation:

- the information of a transaction can be removed when its commit_t < start_t of any uncommitted transaction in the system.



Discussions on Optimistic Method

Serializability

- Transactions are serialized in the order of the time when they are closed.

Starvation problem

- a (long) transaction may be aborted again and again.

Advantage

- optimistic (high concurrency).
- no deadlock.
- a transaction will not be blocked by others.

Disadvantage:

- complicate.
- starvation.
- abort and restart.

Conclusion of Concurrency Control Methods

Two phase locking:
- Lock data items before access.
- Serialized in the order of obtaining locks.
- No abort and restart.

Timestamp ordering:
- Check timestamps of data items before access.
- Serialized in the order of start time.
- Abort during execution.

Optimistic method:
- Validate at the close.
- Serialized in the order of close time.
- Abort at validation.