Atomic transactions 1/4

- Concerned with the problem of providing consistent access to and update of shared data

May have:

- Multiple clients accessing data
- Concurrent accesses
- *Failures* occurring during or after a sequence of accesses to the data
- Clients should be able to take data from one consistent state to another
- Data may be located at a single server/node or distributed over multiple servers/nodes
  - → servers are likely to be multi-threaded

Atomic transactions 2/4

- A transaction is a sequence of operations on shared data that is intended to transform the data from one consistent state to another
  - → e.g. transfer of funds from one bank account to another
- Individual operations may *read* the current value of a data item or *write* (update) the value of a data item
- In a system that supports transactions all operations on shared data must take place in the context of some transaction
  - → classic example is DBMSs where transactions form the normal unit of work
- May have many *simultaneous* transactions

Atomic transactions 3/4

Typically:

- A transaction is started by using a primitive such as "BeginTransaction"
- A transaction executes by reading and writing volatile and persistent data
  - → e.g., database relations, files, objects
- A transaction is ended using primitives such as
  "EndTransaction" to cause the transaction to be *committed*
  "AbortTransaction" to cause the transaction to be *aborted*
- Only the effects of committed transactions are *retained*
- All the effects of aborted transactions are *discarded*
- A transaction may also be aborted due to a failure or by the transaction system in response to abnormal situations

Atomic transactions 4/4

For example, to move IRP100 between two bank accounts:

```
BeginTransaction;
acc1.withdraw(100);
acc2.lodge(100);
EndTransaction;
```

Or:

```
BeginTransaction;
acc1.withdraw(100);
if (acc1.balance( ) < 0)
    AbortTransaction;
acc2.lodge(100);
EndTransaction;
```

Transaction properties

A transaction should have the following properties:

- Atomicity - a transaction's intended changes to the system state either *all* happen or *none* happen
  - → all-or-nothing property; a transaction is an indivisible unit of work
- Consistency - a transaction is a correct transformation of the system state
  - → if the program is logically correct
- Isolation - partial effects of one transaction are not visible to other transactions
  - → also called Independence - transactions execute independently of one another
- Durability - the effects of a committed transaction are permanent

Serialisability 1/5

Consider the following concurrent transactions:

- T1: transfers $10 from acc1 to acc2 T2: transfers $20 from acc3 to acc2

T1	T2	Accounts 1	2	3
BeginTransaction;	BeginTransaction;	100	100	100
balance = acc1.read();		**100**	100	100
acc1.write(balance - 10);		**90**	100	100
	balance = acc3.read();	90	100	**100**
	acc3.write(balance - 20);	90	100	**80**
balance = acc2.read();		90	**100**	80
	balance = acc2.read();	90	**100**	80
acc2.write(balance + 10);		90	**110**	80
EndTransaction;	acc2.write(balance + 20);	90	**120**	80
	EndTransaction;			

Interleaving of transactions causes T1's update to be lost!
"the lost update problem"

Serialisability 2/5

Consider two more concurrent transactions:

- T1: transfers $10 from acc1 to acc2 T2: calculates total of all accounts

T1	T2	Accounts 1	2	3	sum
BeginTransaction;	BeginTransaction;	100	100	100	0
balance = acc1.read();		**100**	100	100	0
acc1.write(balance - 10);		**90**	100	100	0
	sum = acc1.read();	**90**	100	100	**90**
	sum = sum + acc2.read();	90	**100**	100	**190**
balance = acc2.read();		90	**100**	100	190
	sum = sum + acc3.read();	90	100	**100**	**290**
acc2.write(balance + 10);	EndTransaction;	90	**110**	100	290
EndTransaction;					

Interleaving of transactions causes T2 to see *partial effects* of T1
T2 sees an inconsistent (intermediate) state of the system
"the inconsistent retrieval problem"

Serialisability 3/5

Another version of the inconsistent retrieval problem

- T1: transfers $10 from acc1 to acc2 T2: . . .

T1	T2	Account 1
BeginTransaction;	BeginTransaction;	
. . .	. . .	
	balance = acc1.read();	**100**
balance = acc1.read();		100
acc1.write(balance - 10);		90
	balance = acc1.read();	**90**
. . .	. . .	

T2 sees effects of T1 while it is executing leading to an
"unrepeatable read"

Serialisability 4/5

- Notice that these problems occur because the execution of the different transactions is interleaved

T1	T2	Accounts 1	2	3
BeginTransaction;		100	100	100
balance = acc1.read();		**100**	100	100
acc1.write(balance - 10);		**90**	100	100
balance = acc2.read();		90	**100**	100
acc2.write(balance + 10);		90	**110**	100
EndTransaction;	BeginTransaction;	90	110	100
	balance = acc3.read();	90	110	**100**
	acc3.write(balance - 20);	90	110	**80**
	balance = acc2.read();	90	**110**	80
	acc2.write(balance + 20);	90	**130**	80
	EndTransaction;			

If (correct) transactions are executed one after another (i.e., serially) then their combined effects will be consistent

Serialisability 5/5

- Executing transactions serially (one after another) is unnecessarily restrictive
  - → reduces concurrency in the system
- Require that the combined effects of a set of interleaved transactions be *the same as if* the transactions had been executed serially in some order
  - → i.e. the values of all the data items should be the same after the interleaved transactions have completed as they would have been after *some* serial execution of the transactions
- Such an interleaving is said to be *serialisable* or *serially equivalent*
- Serialisability prevents lost updates, inconsistent retrievals, and unrepeatable reads

Concurrency control

- Concurrency control mechanisms are used to ensure serialisability of transactions

There are three basic approaches to concurrency control:

- Locking methods
  - → we will look at these in detail
- Timestamping methods
  - → each transaction is assigned a timestamp and transactions are serialised in timestamp order
  - → a transaction may be too late to access a data item in which case it is aborted
- Optimistic methods
  - → accesses proceed unchecked
  - → before committing, a check is made to see if the current transaction conflicts with any other concurrent transactions; if so some transaction is aborted

Locking 1/4

- Most widely used approach
- A transaction must obtain a read (shared) or write (exclusive) lock on a data item prior to performing a read or write on that data item
- More than one transaction can hold a read lock on the same data item simultaneously
- Only one transaction at a time can hold a write lock on a data item
- A transaction holds a lock until it explicitly releases it
- Effects of the write operation are not visible to other transactions until the write lock has been released

Locking 2/4

- The locking rules are described using a *lock compatibility matrix*

For a single data item:

Held \ Required	Read	Write
None	Yes	Yes
Read	Yes	Wait
Write	Wait	Wait

Lock held by one transaction

Lock required by other transaction

A transaction can *upgrade/promote* its read lock to a write lock if there are no other transactions holding read locks for the same data item

A transaction can *downgrade* a write lock to a read lock

Locking 3/4

- What happens when we use locking in the "lost update" scenario
  - → T1: transfers $10 from acc1 to acc2 T2: transfers $20 from acc3 to acc2

T1	T2	Accounts 1	2	3
BeginTransaction;	BeginTransaction;	100	100	100
read lock acc1;	**read lock acc3;**			
balance = acc1.read();	balance = acc3.read();	**100**	100	**100**
unlock acc1;	**unlock acc3;**			
write lock acc1;	**write lock acc3;**			
acc1.write(balance - 10);	acc3.write(balance - 20);	**90**	100	**80**
unlock acc1;	**unlock acc3;**			
read lock acc2;	**read lock acc2;**			
balance = acc2.read();	balance = acc2.read();	90	**100**	80
unlock acc2;	**unlock acc2;**			
write lock acc2;				
acc2.write(balance + 10);	*<waiting>*	90	**110**	80
unlock acc2;				
EndTransaction;	**write lock acc2;**			
	acc2.write(balance + 20);	90	**120**	80
	unlock acc2;			
	EndTransaction;			

Same result as before !??

Locking 4/4

- Serialisability requires that all of a transaction's access to a single data item be serialised is the same order with respect to other transactions
  - → in particular, all pairs of conflicting operations of two transactions must be serialised in the same order

In our scenario:

- T2's read of account 2 is serialised before T1's write
  - → i.e., as if T2 occurred before T1
- T2's write is serialised after T1's write
  - → i.e., as if T2 occurred after T1

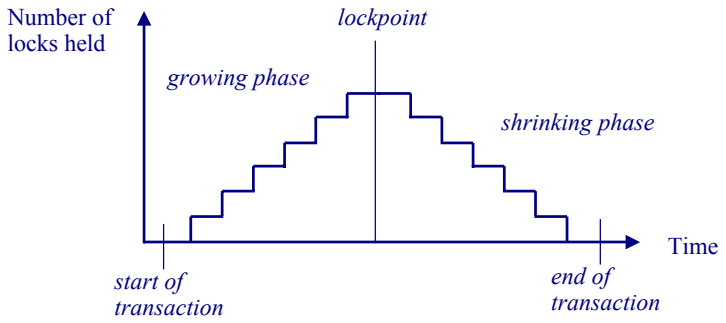
Two-phase locking 1/6

- This requirement can be met using two-phase locking (2PL)

Under 2PL, transactions operate in two distinct phases:

- *Growing phase* during which the transaction acquires locks
- *Shrinking phase* during which it releases those locks
- 2PL ensures that a transaction cannot acquire a lock once it has released any lock
- Read locks may be *upgraded* to write locks during the growing phase
- Write locks may be *downgraded* to read locks during the shrinking phase
- It can be proven that the interleavings produced by transactions that obey 2PL are guaranteed to be serialisable

Two-phase locking 2/6



Two-phase locking 3/6

- Applying 2PL in the "lost update" scenario
  - → T1: transfers $10 from acc1 to acc2 T2: transfers $20 from acc3 to acc2

T1	T2	Accounts 1	2	3
BeginTransaction;	BeginTransaction;	100	100	100
read lock acc1;	**read lock acc3;**			
balance = acc1.read();	balance = acc3.read();	100	100	100
write lock acc1;	**write lock acc3;**			
acc1.write(balance - 10);	acc3.write(balance - 20);	90	100	80
read lock acc2;				
balance = acc2.read();		90	**100**	80
write lock acc2;	*<waiting>*			
acc2.write(balance + 10);		90	**110**	80
unlock acc1,acc2;				
EndTransaction;	**read lock acc2;**			
	balance = acc2.read();	90	**110**	80
	write lock acc2;			
	acc2.write(balance + 20);	90	**130**	80
	unlock acc3,acc2;			
	EndTransaction;			

Two-phase locking 4/6

Consider the following serial execution of T1 and T2 which uses 2PL

- T1: lodges $10 to acc1 T2: lodges $20 to acc1

T1	T2	Account 1
BeginTransaction;	BeginTransaction;	100
read lock acc1;		
balance = acc1.read();		**100**
write lock acc1;	*<waiting>*	
acc1.write(balance + 10);		**110**
unlock acc1;		
	read lock acc1;	
	balance = acc1.read();	**110**
	write lock acc1;	
AbortTransaction;	acc1.write(balance + 20);	**130**
	unlock acc1;	
	EndTransaction;	

Two-phase locking 5/6

- The problem here is that 2PL allows a transaction to see the *uncommitted* effects of another transaction
- If the transaction aborts, the effect is that other transactions may have read values that "never existed"
- Such a read is referred to as a *dirty read*
- Dirty reads violate the isolation property of transactions
- The later transaction becomes *dependent* on the commit of the earlier transaction

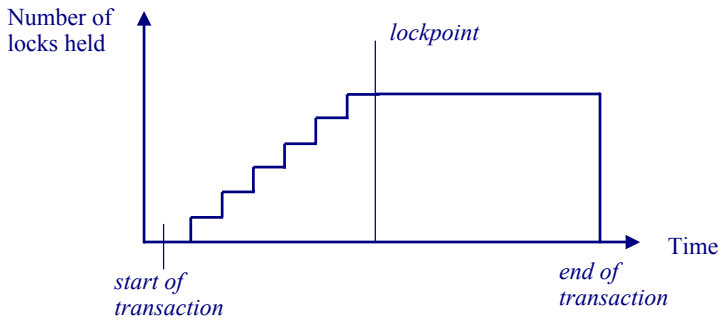
Two-phase locking 6/6

- Could delay committing a transaction until all the transactions on which it depends commit
- This implies aborting a transaction when any transaction on which it depends aborts
- Could lead to *cascading aborts*
- Use group commitment to commit a group of dependent transactions together

Strict two-phase locking 1/3

- Another solution is to *prevent* transactions from reading data items that have been written by uncommitted transactions
- An execution that prevents reads of uncommitted updates is said to be *strict*
- This can be achieved if write locks are held until after the transaction commits or aborts
- Strict two-phase locking is a variant of 2PL that requires all locks to be held until the transaction terminates (commits or aborts)
- Strict 2PL enforces the isolation property and prevents dirty reads

Strict two-phase locking 2/3



Strict two-phase locking 3/3

- Under strict 2PL:

T1	T2	Account 1
BeginTransaction;	BeginTransaction;	100
read lock acc1;		
balance = acc1.read();		**100**
write lock acc1;		
acc1.write(balance + 10);		**110**
AbortTransaction;		**100**
	read lock acc1;	
	balance = acc1.read();	**100**
	write lock acc1;	
	acc1.write(balance + 20);	**120**
	EndTransaction;	

In practice, most systems that use locking use strict 2PL

Deadlock 1/6

Consider the following scenario:

T1	T2	Account 1
BeginTransaction;		100
read lock acc1;	BeginTransaction;	
balance = acc1.read();	**read lock acc1;**	**100**
<T1 wants to update acc1>	balance = acc1.read();	**100**
	<T2 wants to update acc1>	
<waiting>		
	<waiting>	

Here T2 is waiting for T1 to release its read lock
T1 is waiting for T2 to release its read lock
Neither can ever make progress - *deadlock*!

Deadlock 2/6

- Deadlock occurs when a two (or more) transactions are waiting for each other to release locks
  - → T1 waiting for T2 and T2 waiting for T1
  - → T1 waiting for T2, T2 waiting for T3, and T3 waiting for T1
  - → etc

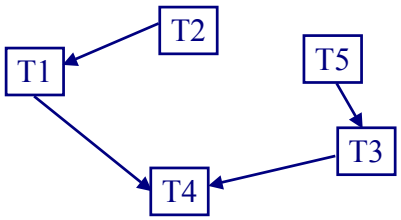
May occur:

- In long-running transactions
  - → interactive transactions
- When certain data items are in frequent use
  - → so-called “hot-spot” data
- Where locks can be promoted
- Anywhere else

Deadlock 3/6

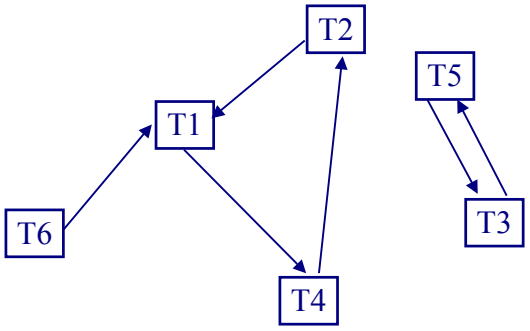
A so-called "*waits-for graph*" (WFG) can be used to to represent waiting relationships between transactions

Each node in the WFG represents a transaction
A directed edge from node T1 to node T2 in the graph means that T1 is waiting for T2 to release a lock



Deadlock 4/6

A *cycle* in the WFG indicates the presence of a deadlock in the system



Deadlock 5/6

Two strategies for dealing with deadlock are possible:

- Deadlock prevention
- Deadlock detection and resolution

Deadlock can be prevented by:

- *Not waiting* when required locks are already held by other transactions in incompatible modes
  - → e.g. abort requesting transaction
- Having transactions acquire all necessary locks when they start
- Locking data items in a pre-defined order
  - → deadlock occurs when T1 has data item A locked and wants to lock data item B while T2 has B locked and wants to lock A
  - → this couldn't happen if A always had to be locked before B

Deadlock 6/6

Deadlock can be detected:

- By maintaining a representation of the WFG as locks are requested
  - → check for cycles whenever a transaction waits (or less frequently)
- By associating timeouts with each lock request
  - → assume a deadlock if lock is not granted within timeout interval and abort requester
- By associating timeouts with locks
  - → having been held for a certain period, a lock becomes vulnerable
  - → an outstanding/new request for a vulnerable lock can be satisfied by reclaiming (breaking) the lock and aborting the previous holder

Deadlock can be resolved by:

- Aborting one of the waiting transactions
  - → choice is non-trivial
  - → want to minimise work lost and maximise chances of deadlock-free execution

Recovery

- Recovery is concerned with ensuring the consistency of the shared data in the presence of *unpredictable* failures
  - → of hardware or software
- Want to restore shared data to a consistent state after any failure
  - → inconsistency may arise as a result of partial updates by interrupted transactions
- Transactions that were interrupted by a failure are *aborted*
- We want the shared data to reflect *all* the modifications due to committed transactions and *none* of the modifications due to aborted transactions
- Recovery is therefore concerned with ensuring the *atomicity* and *durability* properties of transactions in the presence of failure
  - → failure atomicity - no partial updates by aborted transactions
  - → durability - all committed updates survive

Types of failure

There are essentially three types of failure to be considered:

- Transaction failure
- System failure
- Media failure

Transaction failure

- Transaction-induced abort
  - → transaction issues AbortTransaction, e.g., because account is overdrawn after withdrawal
- Unforeseen transaction failure
  - → result of bugs in application (e.g., divide by zero), resource shortages (e.g., out of memory) etc
- System-induced abort
  - → transaction system aborts transaction, e.g., to break a deadlock

In each case:

- No other transactions are affected
- Any updates performed by aborted transaction must be *undone*

System failure

- Machine or operating system crash
  - → e.g. power failure, etc
- Assume partial amnesia crash
  - → contents of volatile memory (including disk buffers) lost
  - → data on secondary storage survives
  - → last write may have been interrupted - disk writes are not atomic (c.f., stable storage)
- Assume fail-silent behaviour

- All on-going transactions aborted
- Any updates performed by aborted transactions must be *undone*
- Updates by committed transactions may need to be *redone*

Media failure

- A failure which results in some part of secondary storage being corrupted
  - e.g., disk head crash, decay, etc
- Must be able to restore last committed value of all data items
- Use archiving or mirroring techniques

Updating shared data

- There are two main strategies for updating shared data items
- Update in place
  - updates applied immediately to the data item on secondary storage
  - be careful not to loose old value in case of subsequent abort
  - make sure new value has actually been made persistent in case of commit
  - record *undo*/*redo* information redundantly
- Shadow versions
  - updates create a new version of the data item
  - on commit, indices are atomically updated to refer to the new version
  - record *redo* information redundantly

Logging 1/3

- Redundant information used for recovery is usually written to a log file
  - → may contain undo and/or redo information depending on update strategy
- A new log record is written each time that
  - → Transaction begins
  - → Transaction updates a data item
    - ◆ Update in place
  - → Transaction commits
  - → Transaction aborts
- Log records for a transaction are written to the log in order
- After a failure, the log can be used to restore the shared data to a consistent state

Logging 2/3

A typical log record might contain:

- The identifier of the transaction concerned
- The type of record
  - → i.e., one of begin, update, commit, abort
- The identifier of the data item affected (for updates)
- The *before image* of the data item (for updates)
  - → i.e., its value before the update
- The *after image* of the data item (for updates)
  - → i.e., its value after the update
- Log management information
  - → e.g., a pointer to the previous log record for the same transaction

Logging 3/3

- Before a data item is changed sufficient recovery information to allow the update to be *undone* if necessary must be logged
  - → before image must be available in case of future failure (including while updating the data)
- Before a transaction commits sufficient recovery information to allow the transaction's updates to be *redone* if necessary must be logged
  - → updates may not have been stored persistently
- Typically use *write-ahead logging* protocol
  - → log records must be written before data items are actually modified - *force writing* if necessary
- All log records for a transaction must be written before the transaction commits
- In particular, a transaction is only committed when its commit record is successfully written to the log

Checkpointing 1/2

- During restart after a failure the log is scanned to determine which transactions have to be redone or undone
- Checkpoints are used to limit how far back in the log the transaction system has to look
  - → the last checkpoint
- Checkpoints are taken periodically

Two approaches are possible

- *asynchronous* checkpoint
  - → taken concurrently with normal processing
- *synchronous* checkpoint
  - → system stops accepting new transactions and waits for all on-going transactions to complete

Checkpointing 2/2

At an asynchronous checkpoint:

- All buffers are force-written to log/secondary storage
- Checkpoint record with list of active transactions is written to the log
  - → address of checkpoint record in the log is written to a special restart file

At a synchronous checkpoint:

- All buffers are force-written to log/secondary storage
- Checkpoint record is written to the log

- A synchronous checkpoint captures a *transaction-consistent system state*
- An asynchronous checkpoint represents a known but not necessarily transaction-consistent system state

Restart and recovery 1/5

- A *cold start* occurs when a system is initialised or restarted from archive after the log or restart file have been corrupted by a catastrophic failure

- A *warm start* occurs after a controlled shutdown of the system
  - → once shutdown is initiated, no new transactions are accepted
  - → once all active transactions have terminated, all buffers are force-written to log/secondary storage
  - → system shuts down

- An *emergency restart* occurs after the system has shut down unexpectedly due to a failure
  - → restart procedure is used to redo effects of committed transactions and undo the effects of aborted transactions as necessary

Restart and recovery 2/5

During restart, the system needs to determine:

- Which transactions had already committed at the time of the failure
  - → redo list
- Which transactions had already aborted at the time of the failure
  - → undo list
- Which transactions were active at the time of the failure
  - → and which are now aborted – undo list

Restart and recovery 3/5

- Redo and undo lists can be determined by scanning the log forward starting from the last checkpoint record
- Given the redo and undo lists, the corresponding log records can be used to redo/undo the effects of the transactions concerned as required
- Restart must be *idempotent*
  - → i.e., repeatable - in case it is itself interrupted by failure
  - → no new transactions are allowed during restart

Restart and recovery 4/5

Restart algorithm for undo/redo logging

```
// Step 1: locate last checkpoint record and get initial
// redo- and undo-lists
read address of last checkpoint from restart file;
read checkpoint record from log;
redo list = empty;
undo list = list of active transactions from checkpoint record;
// Step 2: determine final redo and undo lists
while (not at end of log) {
  read next record from log;
  if (record === begin)
    add transaction to undo-list;
  else if (record == commit)
    move transaction from undo list to redo list;
}
```


Restart and recovery 5/5

```
// step 3: redo/undo updates
while (not at beginning of log) {  // moving backwards
  read next record from log;
  if (record === update && transaction is on undo list)
    undo update using before image from log record;
}
while (not at end of log) {  // moving forwards
  read next record from log;
  if (record === update && transaction is on redo list)
    redo update using after image from log record;
}
```

Distributed transactions 1/2

- Already seen how transactions are implemented on a single node with a combination of appropriate concurrency control and recovery strategies
  - → locking, timestamps, optimistic methods
  - → update in place, shadow versions, logging
- For now we will assume:
  concurrency control is implemented using strict 2PL
  recovery is implemented using in-place update and write-ahead logging
- Want to look at issues that arise when transactions are distributed
  - → i.e. the same transaction may execute at more than one node

Distributed transactions 2/2

- Assume a client/server system
- A single transaction may involve accesses to resources managed by different servers at different nodes
  - → client may make requests to multiple servers
  - → client may make request(s) to one server that uses other servers
- Each server can be made responsible for concurrency control and recovery for the resources that it manages (as in a centralised system)
- *Atomicity* requires that *all* the servers eventually make the same decision as to whether the transaction should commit or abort and act accordingly
  - → despite the failure/temporary unavailability of any server
- Usually allow any server to unilaterally abort a transaction
  - → autonomy; due to failure, local deadlock etc

The distributed commit problem

- The problem of achieving agreement on the outcome of a transaction is known as the *distributed commit problem*
- A protocol that allows such agreement to be reached is known as a *commit protocol*
- *Ideally*, if all the servers agree to commit the transaction, the protocol should commit the transaction
  - → *non-triviality requirement*
- Design a commit protocol that satisfies this constraint taking appropriate account of the possibility of failures occurring during the execution of the protocol, e.g. node failure, lost messages, etc.

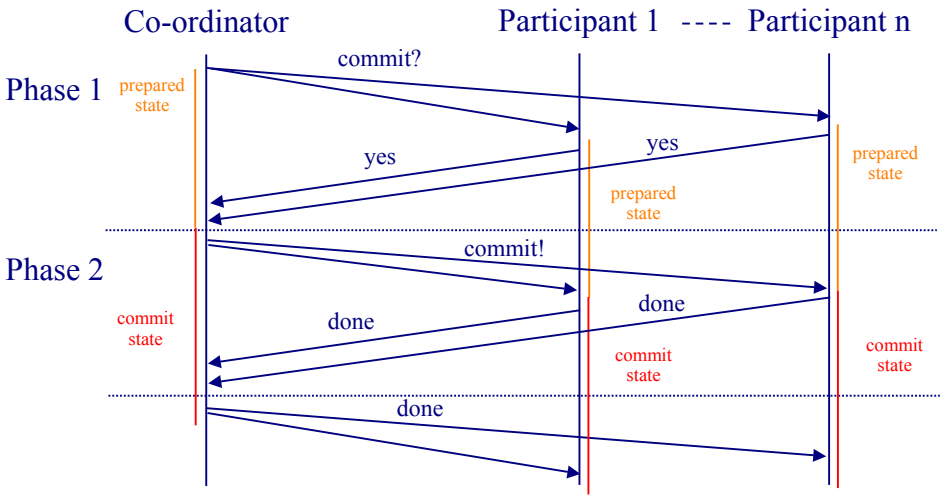
The two-phase commit protocol 1/6

- A number of commit protocols exist
- Of these, the most common is the two-phase commit (2PC) protocol
- The 2PC protocol is executed by all the servers when an attempt to commit a transaction is made

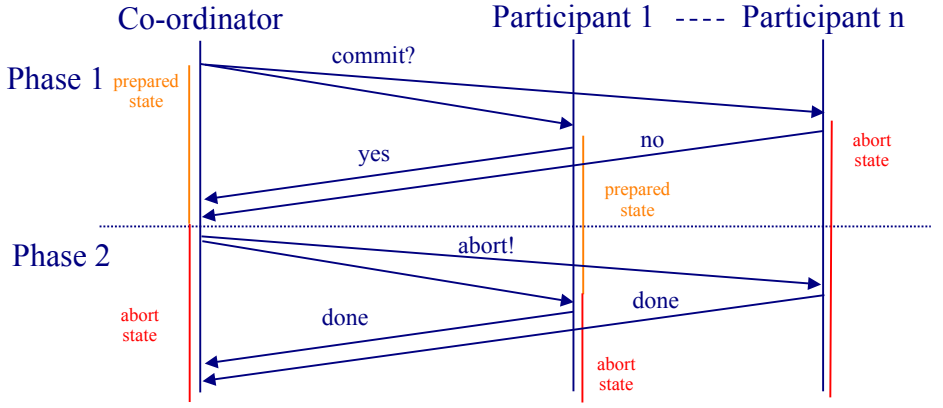
For 2PC:

- One of the servers is designated the *co-ordinator* of the transaction while the other servers are designated as *participants*
- The co-ordinator must know the identities of *all* the participants
- All the participants must know the identity of the co-ordinator
- The commit request is sent to the co-ordinator who proceeds as follows

The two-phase commit protocol 2/6



The two-phase commit protocol 3/6



The two-phase commit protocol 4/6

- When a participant votes yes it can't change its mind later!
  - → loss of autonomy

A participant votes no:

- If it has already decided to abort the transaction
  - → e.g. previous failure, deadlock, etc
- Implicitly by not replying
  - → the co-ordinator will timeout waiting for its reply

The two-phase commit protocol 5/6

Handling failures

- If the commit request message is lost, the co-ordinator will eventually timeout waiting for votes and abort the transaction
- Ditto if any participant fails before voting in phase one
- Ditto if any yes vote is lost
- If a participant fails once in the *prepared* state, it *must* determine the agreed outcome of the transaction when it recovers
- If the commit or abort messages are lost, the co-ordinator will timeout waiting for acknowledgements and retransmit the command
  - → or wait for participants to query status

The two-phase commit protocol 6/6

- Recall that a participant that is in the prepared state must eventually get the outcome of the transaction from the co-ordinator
- If the co-ordinator has failed this could potentially mean waiting for a relatively long time
- The basic 2PC protocol is a *blocking protocol* - commit protocol executions may be *blocked* if the co-ordinator fails
- Note that during this period the resources used by the transaction must remain locked
  - → other transactions may be delayed
- On recovery, a failed co-ordinator will normally (re)transmit the outcome of any on-going commit protocol executions to the participants

Integrating 2PC and recovery 1/5

In phase one:

- The co-ordinator first enters the prepared state:
  - → any information required to commit must be recorded stably in case of a subsequent failure
  - → the co-ordinator writes a `prepared` record to its log including the transaction identifier and the identities of all the participants
- Before a participant votes yes, it enters the prepared state
  - → and writes a `prepared` record to its log giving the transaction identifier and the identity of the co-ordinator
- Before a participant votes no, it enters the abort state
  - → writes an `abort` record to its log

Integrating 2PC and recovery 2/5

In phase two:

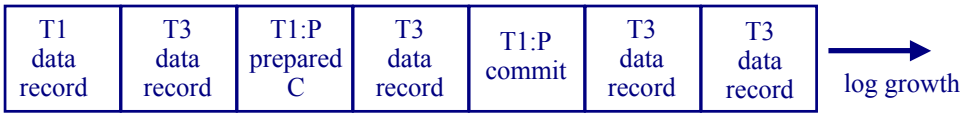
- The co-ordinator writes a `commit` or `abort` record to its log as appropriate
  - → including the transaction identifier
- On receiving the co-ordinator's decision participants write `commit` or `abort` records to their logs as appropriate
  - → assuming that it wasn't done in phase one
- Once it has received acknowledgements from all the participants the co-ordinator writes a `done` record to its log
  - → again including the transaction identifier

Integrating 2PC and recovery 3/5

Example logs

- Co-ordinator:

- Participant:



- On recovery after co-ordinator/participant failure the log is scanned starting at the end
- The recovery actions taken depend on the transaction status at the time of the failure as indicated by the last record for the transaction found in the log

Integrating 2PC and recovery 4/5

Co-ordinator recovery actions

Record	Status/Action
`data`	commit protocol not started; transaction is aborted, abort record written to log and participants informed (or timeout)
`prepared`	no decision had been made; transaction is aborted, abort record written to log and participants informed
`commit`	transaction was committed; inform all participants and resume 2PC protocol
`abort`	transaction was aborted; inform all participants
`done`	2PC protocol was complete; no action required

Integrating 2PC and recovery 5/5

Participant recovery actions

Record	Status/Action
`data`	participant had not yet voted; transaction is aborted
`prepared`	participant failed before it knew co-ordinator's decision; participant requests decision from co-ordinator and acts accordingly
`commit`	transaction was committed; participant sends acknowledgment to co-ordinator allowing it to eventually write `done` record to its log
`abort`	transaction was aborted; no action

Improving the 2PC protocol 1/2

- Question: Can we improve the basic 2PC protocol to avoid blocking? For example, by allowing another participant to take over if the co-ordinator fails?

Improving the 2PC protocol 2/2

- Say that a participant (P), who is prepared and awaiting the outcome to be reported, times out and tries to take over completion of the protocol
- Has the co-ordinator really failed?
- In any case, P can query the other participants and *may* be able to determine the outcome
  - → one of the others may have voted to abort
  - → one of the others may have received a commit or abort message from the co-ordinator
- However, if none of the other participants know the outcome, P has to *wait* for the co-ordinator to recover
  - → co-ordinator may have already decided to commit but not told anybody
  - → or may have told some other participant(s) who didn't reply to P's query
- A *single* failure can still block the protocol

Evaluation of 2PC

- Did we achieve our goal that if all the servers agree to commit the transaction, the protocol should commit the transaction?
- Actually no! Lost messages may cause the protocol to abort a transaction even if all servers are prepared to commit
- If all servers are prepared to commit *and* all their votes reach the co-ordinator then the protocol will commit
- Even amended protocol may block in the presence of co-ordinator failing or becoming partitioned
- $O(n^2)$ messages
- Basic protocol can be applied in other situations where agreement between otherwise autonomous processes must be reached

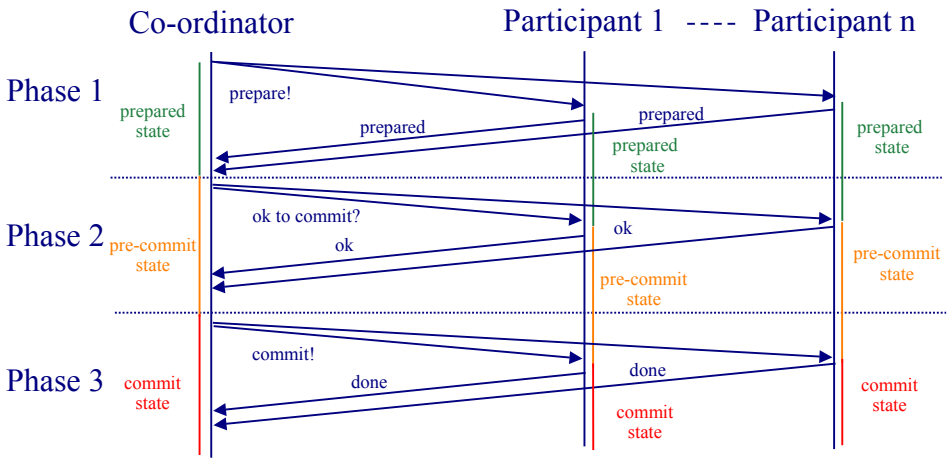
Non-blocking commit protocols

- A commit protocol is *non-blocking* if it never requires an operational participant to block
  - → even if any other participant fails
- A participant that remains operational will always be to able drive the protocol forward to commit or abort in such a way that all other participants will eventually come to the same decision
- Design a non-blocking commit protocol
  - → Hint: may need to impose some additional constraints

The three-phase commit protocol 1/10

- Best known non-blocking commit protocol is the three-phase commit (3PC) protocol
- 3PC is non-blocking provided that *only* fail-silent failures occur
  - → i.e., no lost messages/network partitions
  - → failures are detectable
- Basically the protocol ensures that the state of the protocol execution can always be deduced by the participants that remain operational
  - → provided that they can communicate with each other

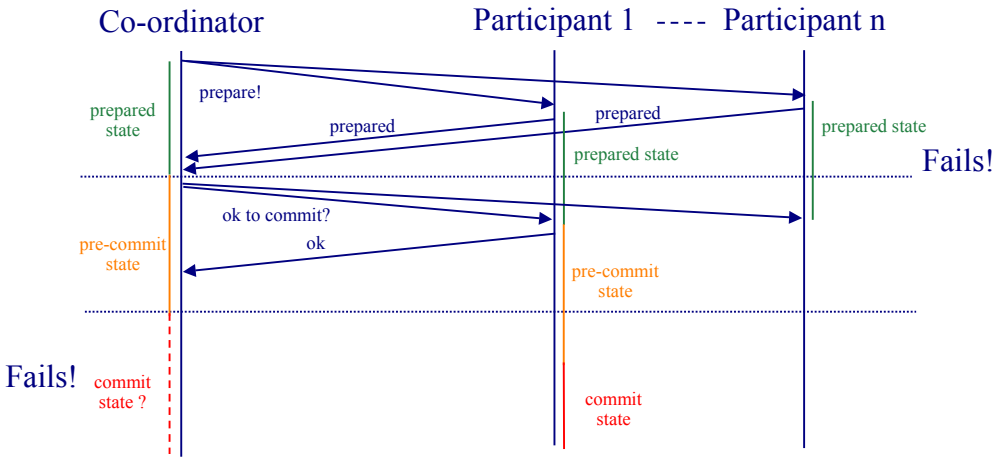
The three-phase commit protocol 2/10



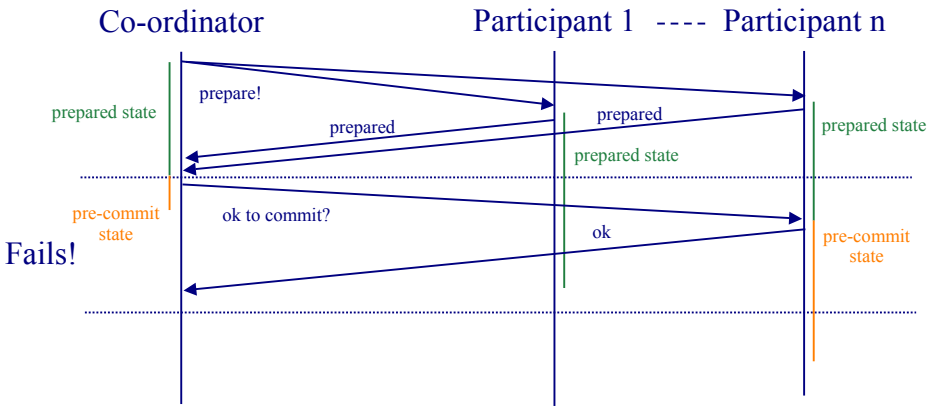
The two-phase commit protocol 3/10

- When a participant votes prepared it can't change its mind later!
  - → loss of autonomy
- If any participant votes prepared and then fails, it must run a recovery protocol to determine the outcome of the protocol when it restarts
- So, in phase two only the *operational* participants need to agree in order for the co-ordinator to commit
  - → commit only if all operational participants are in pre-commit state
- If the co-ordinator fails and all operational participants are pre-committed can still commit
- In 3PC a participant in the pre-commit state knows that *all* the other participants are at least in the prepared state
- A participant in the prepared state knows that *no* other participant has committed

The three-phase commit protocol 4/10

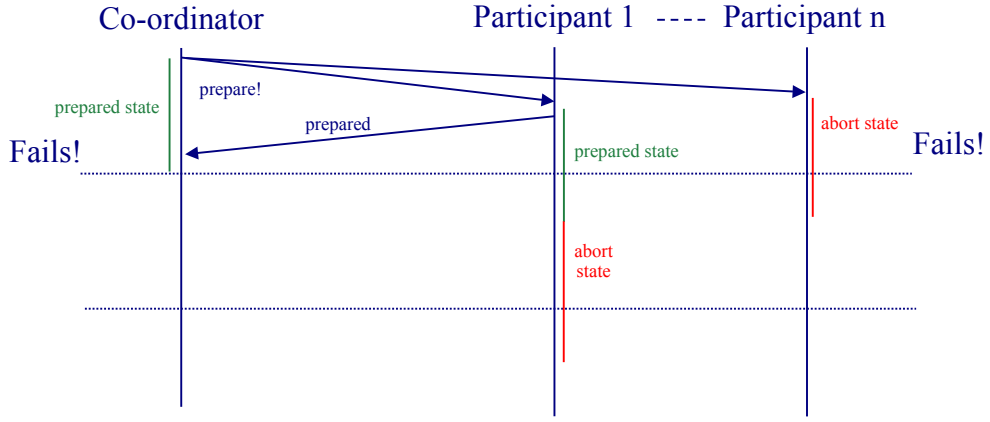


The three-phase commit protocol 5/10



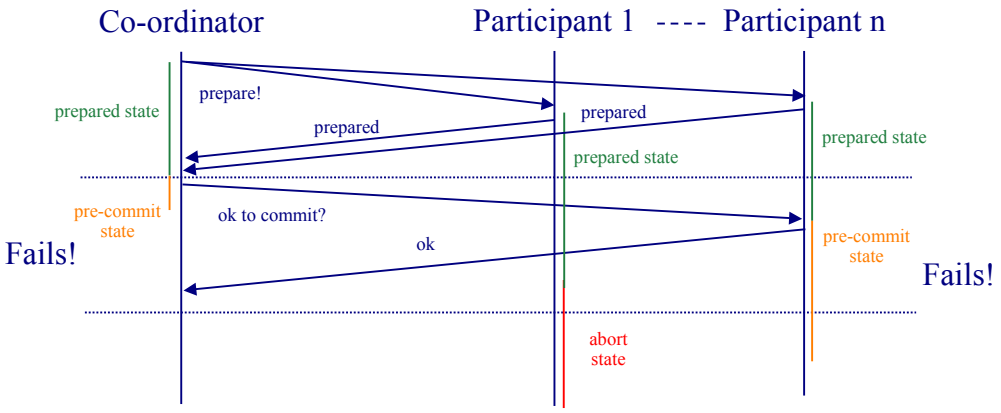
Here we have operational participants in both prepared and pre-commit states
Since Pn is pre-committed, everyone was prepared - commit

The three-phase commit protocol 6/10



Abort if all operational participants are in the prepared state

The three-phase commit protocol 7/10



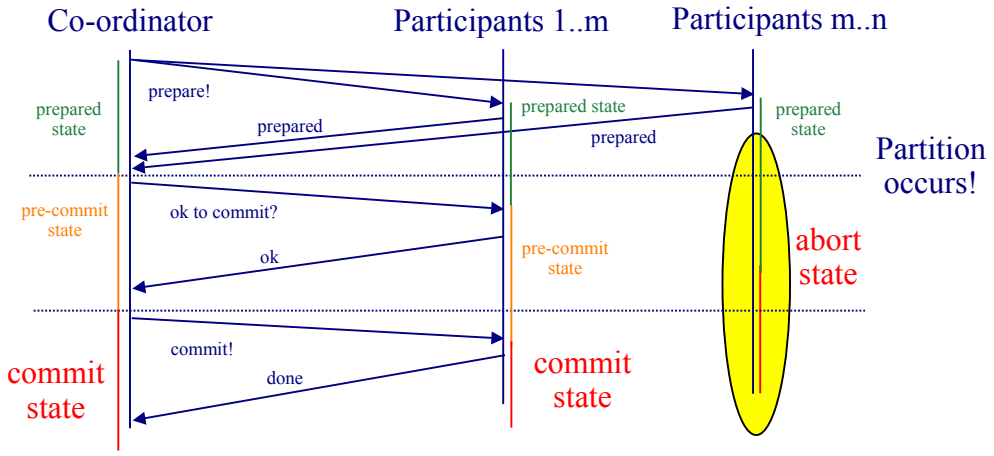
P1 can be sure no one has committed since it hasn't pre-committed!

The three-phase commit protocol 8/10

- On restart, a recovering process has to find an operational process in order to determine the outcome
- Worst case is if all the processes have failed
- Recovering processes need to find one of the *last* processes to have failed
  - → may have to wait for that process to recover

The three-phase commit protocol 9/10

- What happens if we admit the possibility of lost messages/partitions?



The three-phase commit protocol 10/10

- It can be shown that if the communication network can fail such that two operational participants cannot communicate with each other no protocol can be non-blocking

Aside: The FLP impossibility result

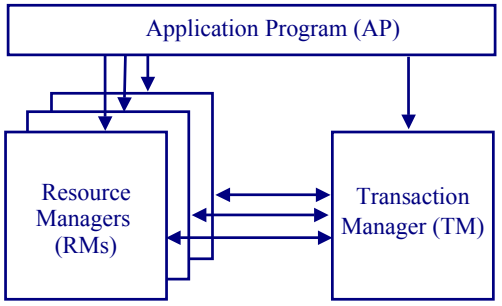
See [Fischer et al. 1985]

- In 1985 Fischer, Lynch, and Paterson *proved* that in an asynchronous distributed system consensus was impossible in the presence of at least one faulty process
- An asynchronous distributed system is characterised by the absence of any concept of time
  - → message passing is reliable but may suffer unbounded delays
  - → processes may fail only by crashing
- Impossible means "not always possible"
  - → all protocols have the "possibility of nontermination" [Fischer et al. 1985]
- Intuition is that a message that is critical to establishing consensus might be delayed indefinitely

Open transaction processing

- Assumed that every server is able to execute the commit protocol
- In practice it may be preferable to separate resource-specific aspects such as concurrency control and recovery from global transaction management
- Make each server responsible for resource-specific actions
- Introduce a Transaction Manager
  - → responsible for transaction management
- This approach has been standardised in the X/OPEN Distributed Transaction Processing (DTP) model
- Allows different types of server (from different vendors) to interwork in one transaction
  - → also different Transaction Managers

The X/OPEN DTP model 1/3



- One TM per node
- An RM can manage any sort of resource
  - → RM could be a DBMS, file server, object server, name server etc
- Transactions can access different types of resources consistently

The X/OPEN DTP model 2/3

- Each AP defines transaction boundaries via the TM interface
- Each AP can use resources managed by a set of RMs
- The TM and RMs exchange transaction control information

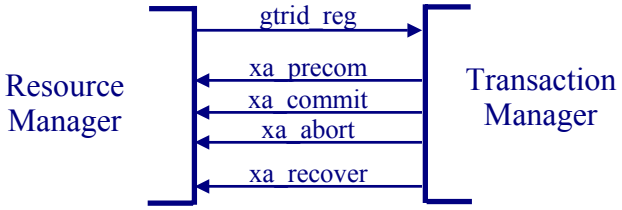
- X/OPEN defines:
  the interface between an AP and the TM - the TX-interface
  the interface between a RM and the TM - the XA-interface

- The interface between an AP and an RM is RM-specific
  - → e..g SQL, file handling system calls etc

- The interface between TMs is defined by an OSI standard

The X/OPEN DTP model 3/3

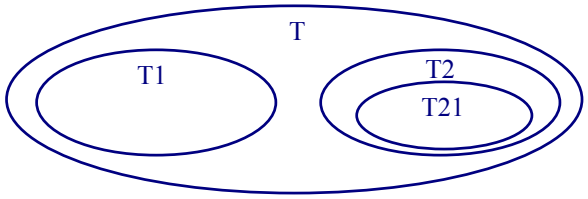
- The XA-interface



- `gtrid_reg` registers an RM as a participant in a transaction
- `xa_precom` requests an RM to prepare - RM may, of course, refuse
- `xa_commit`, `xa_abort` allow the TM to inform the RM of the outcome of the transaction
- `xa_recover` is called during restart to allow the TM to get a list of transactions which are prepared at the RM

Nested transactions

- Nesting allows a transaction to create further transactions
- A *top-level transaction* can create a number of *sub-transactions*
- Sub-transactions can themselves create further sub-transactions
  - → hierarchy of transactions



- T is top-level transaction and parent of T1 and T2; T21 is child of T2; descendants of T are T1, T2 and T21; ancestors of T21 are T2 and T

Advantages of nested transactions 1/2

- Concurrency
  - sub-transactions at the same level in the hierarchy may run concurrently
    - additionally some systems allows a sub-transaction to run concurrently with its parent
    - useful in a distributed system if parts of transaction can run at different nodes
- Independent failure
  - part of a transaction can fail without causing the entire transaction to abort
    - very useful in a distributed system
  - the abort of a sub-transaction does not cause its parent to abort
  - the commit of a sub-transaction is *dependent* on the commit of its parent
    - abort of parent causes abort of child
  - forward error recovery is possible
    - parent can react to failure of child
- Modularity
  - effect of executing BeginTransaction inside a transaction?

Advantages of nested transactions 2/2

Examples:

- Consider posting a mail to a mailing list as a transaction
- No nesting: if mail cannot be delivered to all recipients posting will fail
- Nesting: use a sub-transaction to post message to each intended recipient
  - top-level transaction can decide how to handle individual failures
  - abort whole mail or send to as many recipients as possible
- Bank might use a transaction to process all the standing orders for a day
- If this transaction is structured as a collection of sub-transactions (one per standing order), failure of individual orders can be tolerated

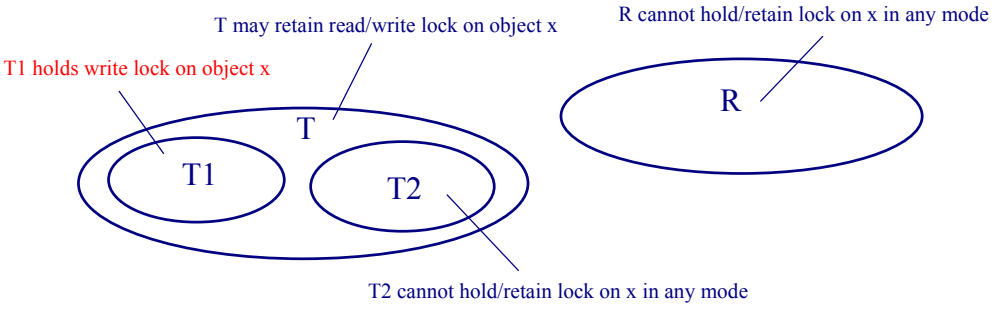
Concurrency control for nested transactions

- Assuming that we are using locking, we need to modify the locking rules to cater for nested transactions
- Essentially (concurrent) sub-transactions at the same level are serialised
  - → as well as top-level transactions
- A sub-transaction may acquire locks held by its ancestors
  - → in competition with its peers
- Locks acquired by a sub-transaction are *inherited* by its parent if the child commits
- Such locks are *retained* until the top-level transaction commits (as usual)

Concurrency control for nested transactions

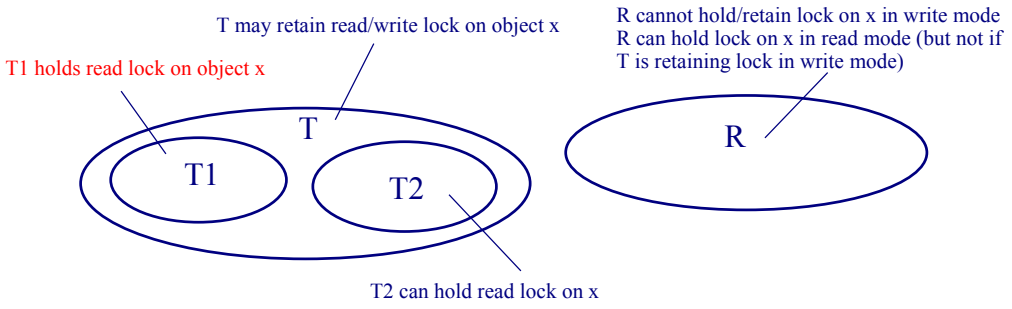
Locking rules

1. A (sub-)transaction may *hold* a lock in write mode if no other (sub-) transaction holds the lock (in any mode) and all *retainers* of the lock are ancestors of the (sub-)transaction



Concurrency control for nested transactions

2. A (sub-)transaction may hold a lock in read mode if no other (sub-) transaction holds the lock in write mode and all retainers of the lock in write mode are ancestors of the (sub-)transaction



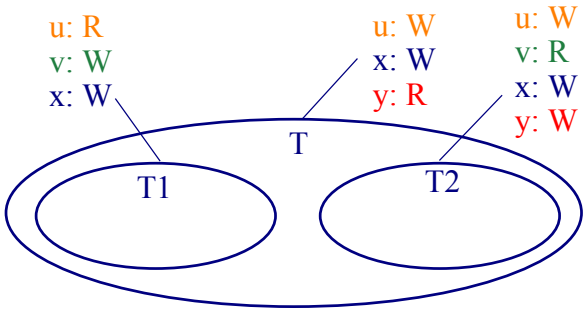
Concurrency control for nested transactions

3. When a (sub-)transaction aborts, all of its locks (held or retained, of all modes) are simply discarded. If any of its ancestors hold or retain the same lock, they continue to do so in the same mode as before the abort

4. When a transaction commits, all of its locks (held or retained, of all modes) are inherited by its parent (if any) in the same mode as the child held or retained them
   → union

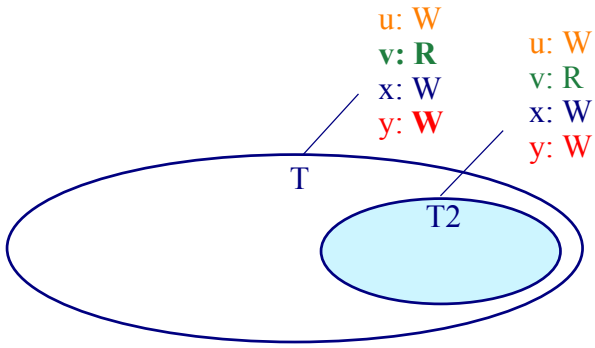
Concurrency control for nested transactions

- Consider the locks *used* by transaction T and its sub-transactions



Assume that T1 ran before T2. If T1aborts and T2 (tentatively) commits ...

Concurrency control for nested transactions



Recovery for nested transactions

- Logically each (sub-)transaction that acquires a write lock on a resource creates a new tentative version of the resource
  - → initialised from the committed version or its parents version
  - → logically have a *stack* of versions
- The (sub)transaction modifies its own tentative version
- If the (sub)transaction aborts, its version is discarded with its write lock
- If the (sub)transaction commits, its version is inherited by its parent and replaces its parent's version (again along with its write lock)
- Reads are always directed to the most recent version created by an ancestor
  - → or the committed version if no version created by ancestors
- Only final version needs to be made permanent during commit of top-level transaction

2PC protocol for nested transactions

- Only execute 2PC protocol when top-level transaction tries to commit
- Sub-transactions may have tentatively committed or aborted at several nodes
- Nodes where sub-transactions tentatively committed may since have failed
- Top-level transaction builds up a list of tentatively committed and aborted sub-transactions
- Each participant is asked to prepare any tentatively committed transactions that are to be committed
  - → not necessarily all those that were tentatively committed at its node
- Operation of 2PC is otherwise as before

References

- [Fischer et al. 1985] Michael Fischer, Nancy Lynch, and Michael Paterson, *Impossibility of Distributed Consensus with One Faulty Process*, JACM 32(2):374-382, April 1985.

See also:

- [Gray and Reuter 1993] Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993, ISBN 1-55860-190-2.
- [Bernstein et al.1987] Philip Bernstein et al., *Concurrency Control and Recovery in Distributed Systems*, Addison-Wesley, 1987, ISBN 0-201-10715-5
- [Birman 1996] Ken Birman, *Building Secure and Reliable Network Applications*, Manning/Prentice Hall, 1996, ISBN 1-884777-29-5, Chapter 13.