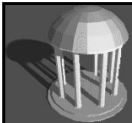


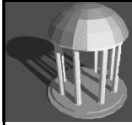
COMP 190 -- Transactions



Example – Balance Transfer

```
.....  
savings = Read(savings_account)  
If savings < 100  
    exit(ERROR_INSUFFICIENT_FUNDS)  
savings = savings - 100  
Write(savings_account, savings)  
checking = Read(checking_account)  
checking = checking + 100  
Write(checking_account, checking)
```

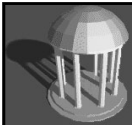
```
.....
```



Example – Credit Interest

```
.....  
savings = Read(savings_account)  
savings = savings * 1.004375  
Write(savings_account, savings)
```

```
.....
```

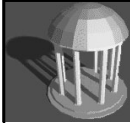


Example – Balance Transfer

```
.....  
savings = Read(savings_account)  
If savings < 100  
    exit(ERROR_INSUFFICIENT_FUNDS)  
savings = savings - 100  
Write(savings_account, savings)  
checking = Read(checking_account)  
checking = checking + 100
```



```
.....
```

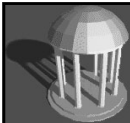


Example – Concurrent Execution

```
.....
savings = Read(savings_account)
savings = savings * 1.004375
.....
savings = Read(savings_account)
If savings < 100
  exit(ERROR_INSUFFICIENT_FUNDS)
savings = savings - 100
Write(savings_account, savings)
checking = Read(checking_account);
checking = checking + 100
Write(checking_account)
.....
Write(savings_account, savings)
```

COMP 190 -- Fall 2001

5

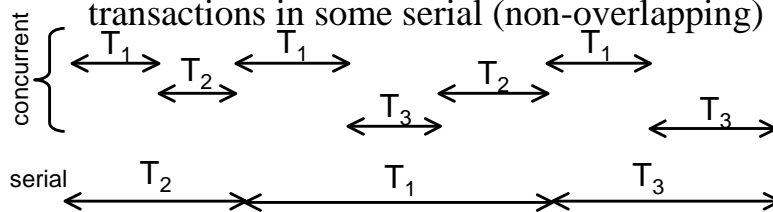


Transaction Properties

- ◆ A programming abstraction that provides for correct operation in the presence of concurrent executions and failures.

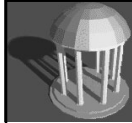
- ◆ Property 1: *Isolation (Serializable)*

The concurrent execution of multiple transactions produces the same result as executing those transactions in some serial (non-overlapping) order.



COMP 190 -- Fall 2001

6



Transaction Properties

◆ Property 2: ***Atomicity***

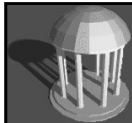
If a transaction is not completed because of some failure (application, system, or hardware), partially completed results will be undone.

◆ Property 3: ***Durability(Permanence)***

If a transaction is completed successfully, the results will (almost certainly) not be lost.

◆ Property 4: ***Consistency***

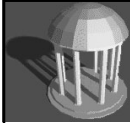
We assume that transaction programs are written to preserve the application's notion of consistency in the data they use.



Transaction Properties

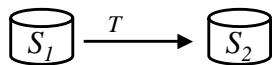
◆ These properties are expressed in the mnemonic ***ACID***

Atomic,
Consistent,
Isolation,
Durable

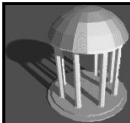


Consistency and Serializability

- ◆ A **correct** transaction, T , transforms a consistent state of the system, S_1 , into another consistent state, S_2 (intermediate states that are inconsistent are allowed if atomicity is enforced).



- ◆ A serial execution of correct transactions preserves consistency.



Consistency and Serializability

In a serial execution of transactions, all operations of one transaction, T_i , complete before any operation of another, T_j , begin. We use the following notation to indicate that T_i “happens before” T_j $T_i \rightarrow T_j$

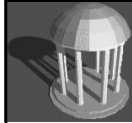
If T_i and T_j are correct, no interference is possible (and therefore consistency is preserved) if either $T_i \rightarrow T_j$ or $T_j \rightarrow T_i$

If the concurrent execution of T_p, T_j, T_k, T_l , is **serializable**, it produces the same effect as some serial execution, e.g.,

$$T_j \rightarrow T_i \rightarrow T_l \rightarrow T_k$$

Therefore, serializable executions of correct transactions maintain consistent states





Transaction Programming

◆ The programming interface consists of:

$T = \text{Begin}();$

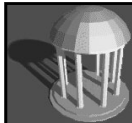
- marks the beginning of a transaction scope; returns a transaction identifier, T

$\text{Commit}(T);$

- marks the end of a transaction scope; makes all results permanent

$\text{Abort}(T);$

- marks the end of a transaction scope; obliterates the effects of the transaction everywhere



Some Notation

$r_i[x] = \text{Read}(x)$ by transaction T_i

$w_i[x,v] = \text{Write}(x,v)$ by transaction T_i

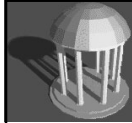
$w_i[x] = \text{Write}(x,-)$ by transaction T_i (when v unspecified)

$c_i = \text{Commit}$ by transaction T_i

$a_i = \text{Abort}$ by transaction T_i

A *history* is a sequence of such operations, in the order that the transaction system processed them.

We assume that Read and Write are the lowest level indivisible (atomic) operations supported.



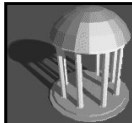
Equivalent Histories

Read and Write requests by a transaction, T_i **conflict** with Read and Write requests by T_k on $[x]$ for **non-commuting** operations, i.e.,

$$\{r_j[x], w_k[x]\} \{w_i[x], r_k[x]\} \{w_i[x], w_k[x]\}$$

Two histories are *equivalent* if they have the same operations and all conflicting operations are in the same order in both histories.

- only the relative order of conflicting operations can influence how the history alters the state of data



Serializable Histories

- ◆ A history is serializable if it is equivalent to a serial history

- ◆ For example,

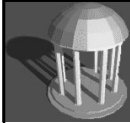
$$H_1 = r_1[x] r_2[x] w_1[x] c_1 w_2[y] c_2$$

is equivalent to

$$H_4 = r_2[x] w_2[y] c_2 r_1[x] w_1[x] c_1$$

($r_2[x]$ and $w_1[x]$ ordered the same in H_1 and H_4 .)

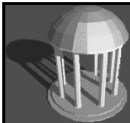
- ◆ Therefore, H_1 is serializable.



Example of Serialization

$T_1 = \text{Begin}();$	$T_2 = \text{Begin}();$
$A = \text{Read}(x);$	$B = \text{Read}(x);$
$A = A + 1;$	$B = B + 1;$
$\text{Write}(x, A);$	$\text{Write}(y, B);$
$\text{Commit}(T_1);$	$\text{Commit}(T_2);$

- ◆ $H = r_1[x] r_2[x] w_1[x] c_1 w_2[y] c_2$
- ◆ H is equivalent to executing T_2 followed by T_1
- ◆ T_1 started and finished before T_2 , yet the effect is that T_2 ran first.

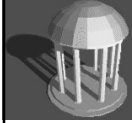


Non-Serializable Example

$T_1 = \text{Begin}();$	$T_2 = \text{Begin}();$
$A = \text{Read}(x);$	$B = \text{Read}(x);$
$A = A + 100;$	$B = B + 100;$
$\text{Write}(x, A);$	$\text{Write}(x, B);$
$\text{Commit}(T_1);$	$\text{Commit}(T_2);$

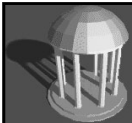
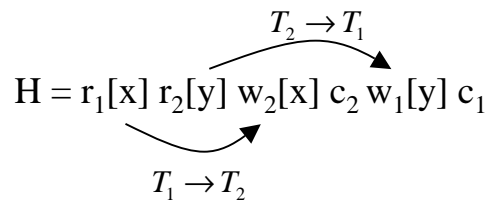
$H = r_1[x] r_2[x] w_2[x] w_1[x] c_1 c_2$ (*lost update*)

$\xrightarrow{T_2 \rightarrow T_1}$
 $\xleftarrow{T_1 \rightarrow T_2}$



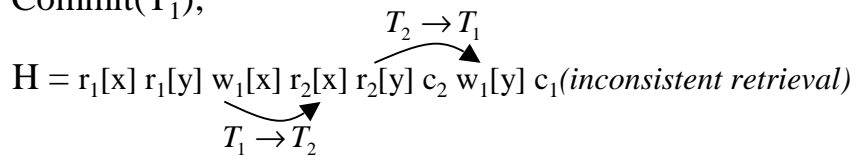
Non-Serializable Example

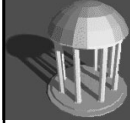
$T_1 = \text{Begin}();$	$T_2 = \text{Begin}();$
$A = \text{Read}(x);$	$B = \text{Read}(y);$
$\text{Write}(y, A);$	$\text{Write}(x, B);$
$\text{Commit}(T_1);$	$\text{Commit}(T_2);$



Non-Serializable Example

$T_1 = \text{Begin}();$	$T_2 = \text{Begin}();$
$A = \text{Read}(x);$	$A = \text{Read}(x);$
$B = \text{Read}(y);$	$B = \text{Read}(y);$
$A = A - 100;$	$\text{Sum} = A + B;$
$B = B + 100;$	$\text{Print}(\text{Sum});$
$\text{Write}(x, A);$	$\text{Commit}(T_2);$
$\text{Write}(y, B);$	
$\text{Commit}(T_1);$	





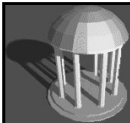
Implementing Serializability

$rl_i[x]$ = Lock x for *reading* by transaction T_i

$wl_i[x]$ = Lock x for *writing* by transaction T_i

$ru_i[x]$ = Unlock read lock on x by transaction T_i

$wu_i[x]$ = Unlock write lock on x by transaction
(Lock and Unlock operations must be atomic.)



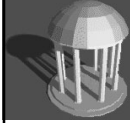
Basic Locking

Lock requests from a transaction, T_i *conflict* with locks held by T_k on $[x]$ for *non-commuting* operations, i.e.,

$\{rl_i[x], wl_k[x]\}$ $\{wl_i[x], rl_k[x]\}$ $\{wl_i[x], wl_k[x]\}$

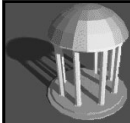
$wl_1[x]$ $w_1[x]$ $rl_2[x]$ $r_2[x]$ (*conflicting*)

$wl_1[x]$ $w_1[x]$ $wu_1[x]$ $rl_2[x]$ $r_2[x]$ (*not conflicting*)



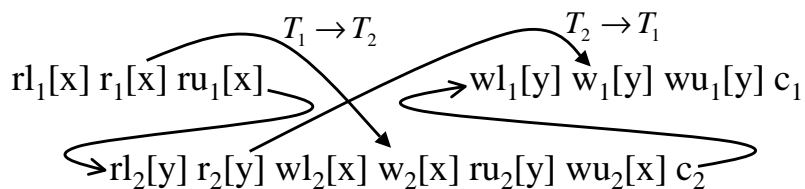
Two-Phase Locking Rules

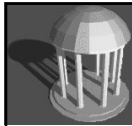
1. If T_j requests a lock on $[x]$ that conflicts with a lock held by T_i , then T_j is forced to wait for an unlock on $[x]$ by T_i
(prevents two transactions from using conflicting operations)
2. Once a lock is granted, it cannot be released by the system until the corresponding read or write operation has been completed.
(ensures operations are processed in lock order)
3. Once any lock for T_j has been released, T_j may not obtain any more locks.
(guarantees that all pairs of conflicting operations from two transactions are scheduled in the same order)



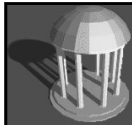
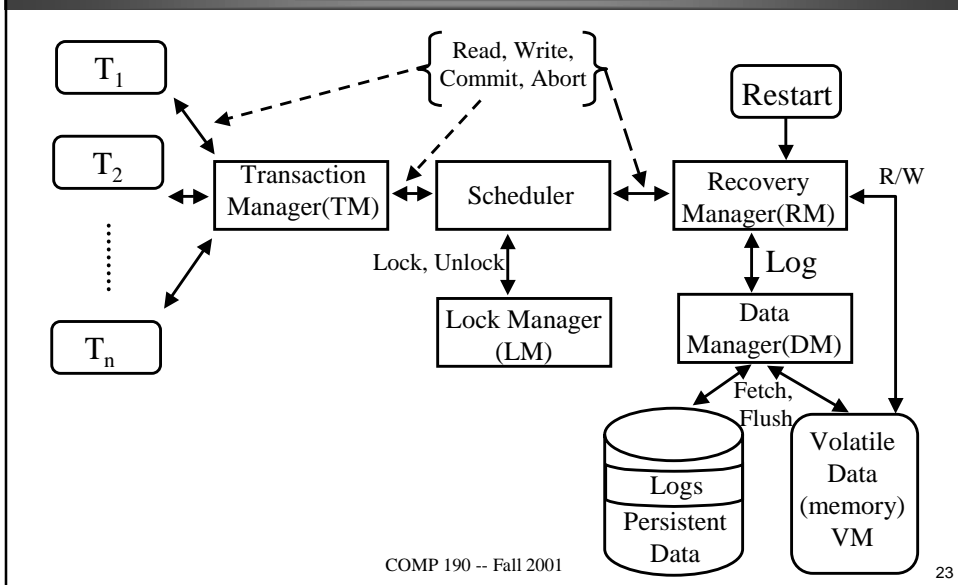
Example Without Rule 3

$T_1 = \text{Begin}();$	$T_2 = \text{Begin}();$
$A = \text{Read}(x);$	$B = \text{Read}(y);$
$\text{Write}(y, A);$	$\text{Write}(x, B);$
$\text{Commit}(T_1);$	$\text{Commit}(T_2);$





Transaction System Structure

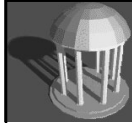


The Recovery Manager

The Recovery Manager (RM) ensures that the system state contains all the effects of committed transactions and none of the effects of uncommitted (or aborted) transactions.

The RM implements the abstraction of durability and recoverability in case of aborts and failures.

If the RM detects any condition that would not preserve durability for a transaction that attempts to commit, it forces the transaction to abort. The RM has responsibility for the final decision on committing (or aborting) a transaction that calls `Commit()`.



Recoverability

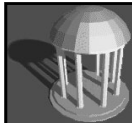
To abort T_i , the RM must wipe out any effects from the execution of the transaction by:

- “undoing” any writes by T_i
- aborting any transaction, T_j , that read data written by T_i (this is called a *cascading* abort)

Example:

$H = w_1[x] r_2[x] w_2[y] a_1$

to abort T_1 , undo $w_1[x]$ and abort T_2 (may cascade further)



Recoverability

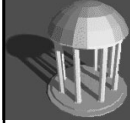
What if the history were the following?

$H = w_1[x] r_2[x] w_2[y] c_2 a_1$

This would imply the necessity to abort T_2 and restore the previous values of both $[x]$ and $[y]$. However, T_2 has already been committed and its effect on $[y]$ should be *permanent* -- aborting T_2 would be a violation of the transaction guarantees!

The *recoverability rule*: the commit of a transaction, T_i , must happen *after* the commits of all transactions that wrote data values read by T_i . A recoverable history is:

$H = w_1[x] r_2[x] w_2[y] c_1 c_2$



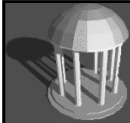
Avoiding the Cascading Aborts

The *anti-cascading rule*: all reads by a transaction, T_i , must happen *after* the commits of all transactions that wrote data values read by T_i . A history that avoids cascading aborts is:

$$H = w_1[x] c_1 r_2[x] w_2[y] c_2$$

while the following history includes a cascading abort:

$$H = w_1[x] r_2[x] w_2[y] a_1$$



Undo With “Before Images”

The easiest way to undo a write, $w[x,v]$, is to restore its *before image*, the value of x *before* $w[x,v]$ executed.

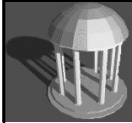
For example, $H = w_1[x,1] w_1[y,3] c_1 w_2[y,1] r_2[x] a_2$

↖
before image of
 $w_2[y,1] = 3$

To implement a_2 , the system can execute $w_s[y,3]$

But this is not always possible without additional rules, e.g.,

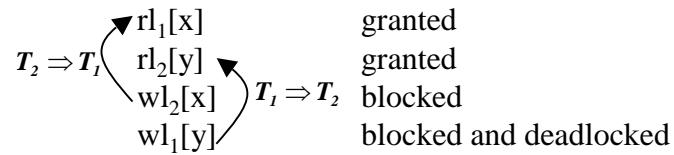
$w_1[x,2] w_2[x,3] a_1 a_2$
↖ before image of $w_1[x,2] = 0$
↖ before image of $w_2[x,3] = 2$



Locking and Deadlocks

Two-Phase Locking (strict or not) is subject to deadlocks.

Example



Explicit deadlock detection - Use a *Waits-For Graph*

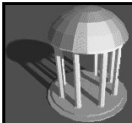
Nodes = {transactions}

Edges = $\{T_i \Rightarrow T_k \mid T_i \text{ is waiting for } T_k \text{ to release a lock}\}$

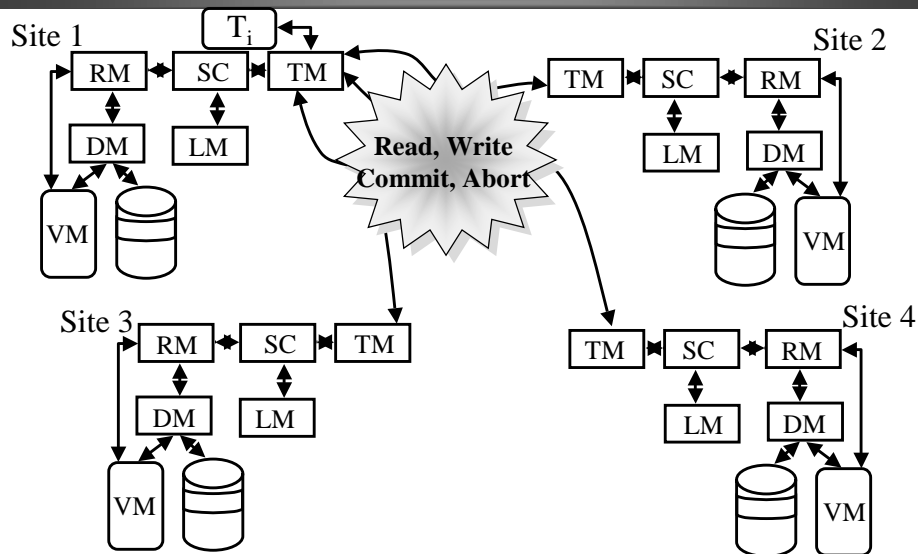
in above example $T_2 \Rightarrow T_1 \Rightarrow T_2$

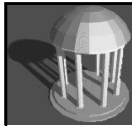
A deadlock is detected by finding a cycle in the waits-for graph.

Choose one transaction (which one?) in the cycle and abort it!

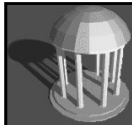
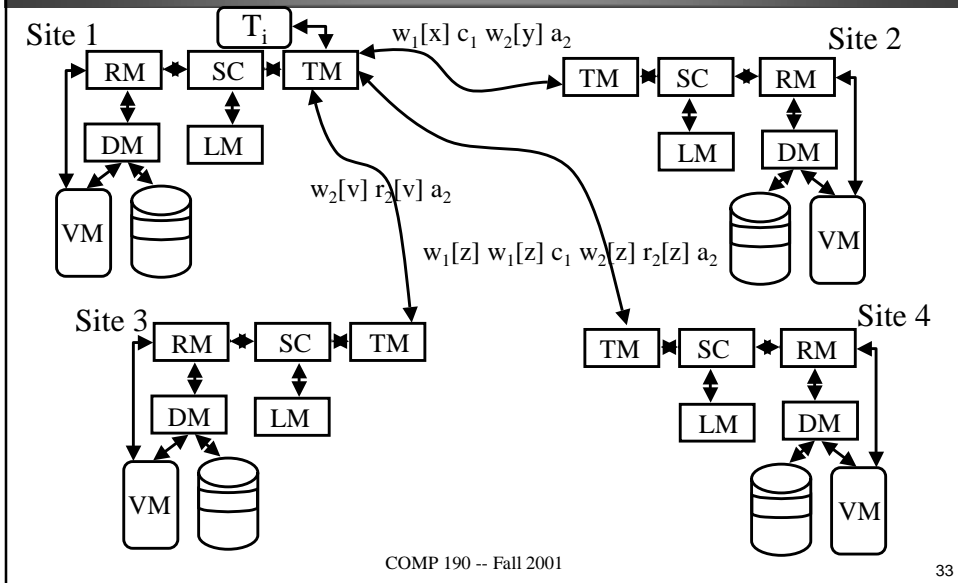


Distributed Transactions





Distributed Transactions

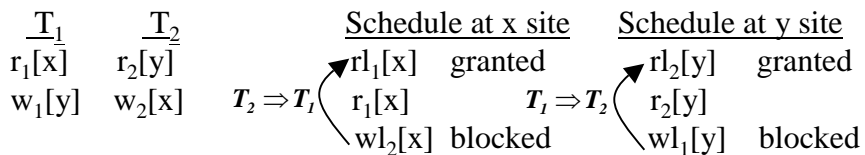


Distributed Two-Phase Locking

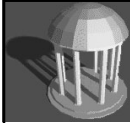
Each site's scheduler manages locks with two-phase rules

Each site's scheduler cannot release locks for T_i until it knows that T_i will not make any more lock requests at any site (when does it know this?).

Deadlocks may now be *distributed deadlocks*



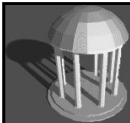
Requires a "global" deadlock detection algorithm



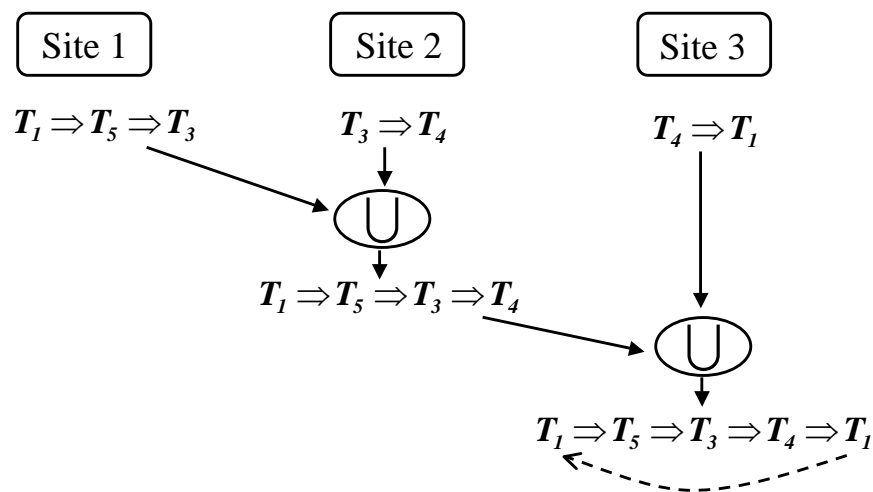
Global Deadlock Detection

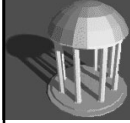
Centralized Strategy: each site send its local waits-for graph to a server that computes the graph unions and checks for cycles.

Distributed Strategy (Path Pushing): each site computes all paths in the local waits-for graph and propagates them to other sites. The receiving sites compute the union with other known paths, check for cycles, and propagate new paths to other sites.



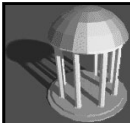
Deadlock Detection with Path-Pushing



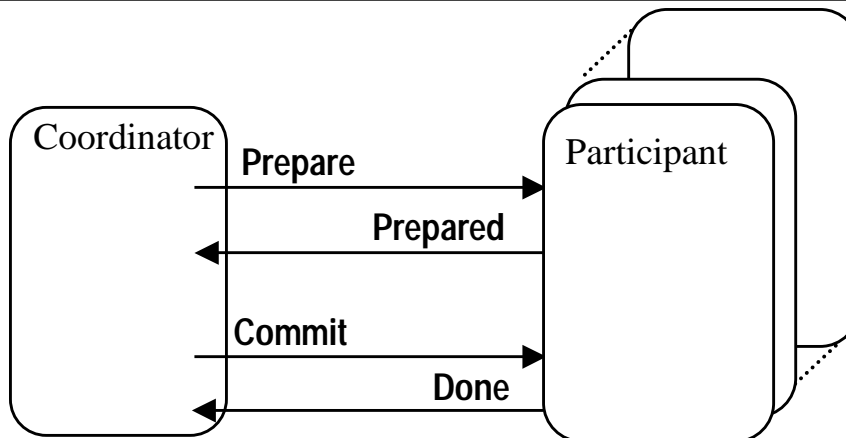


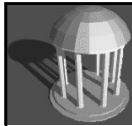
Distributed Commit/Abort Decisions

- ◆ Given that transaction T_i accesses data controlled by RM components at multiple sites (RM_1, RM_2, \dots, RM_n)
- ◆ The requirements are:
 - commit T_i at all of RM_1, RM_2, \dots, RM_n , *or*
 - abort T_i at all of RM_1, RM_2, \dots, RM_n , *and*
 - do so even if an RM fails, a site crashes, a network failure isolates some sites, etc.
- ◆ Never commit at RM_i and abort at RM_k

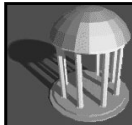
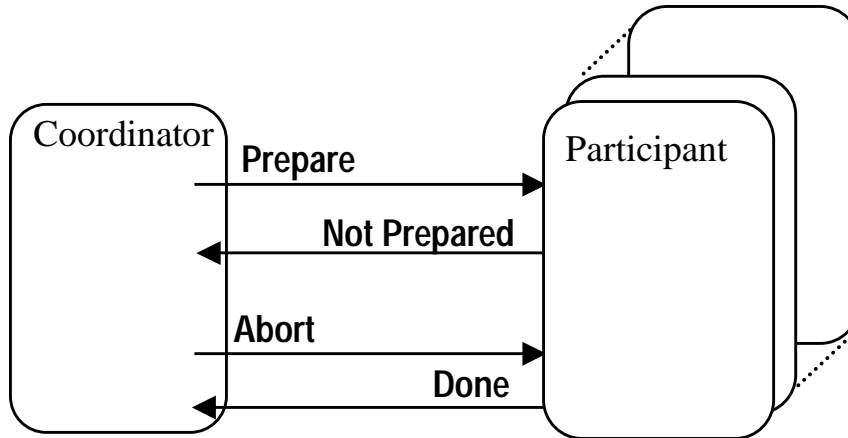


Two-Phase Commit

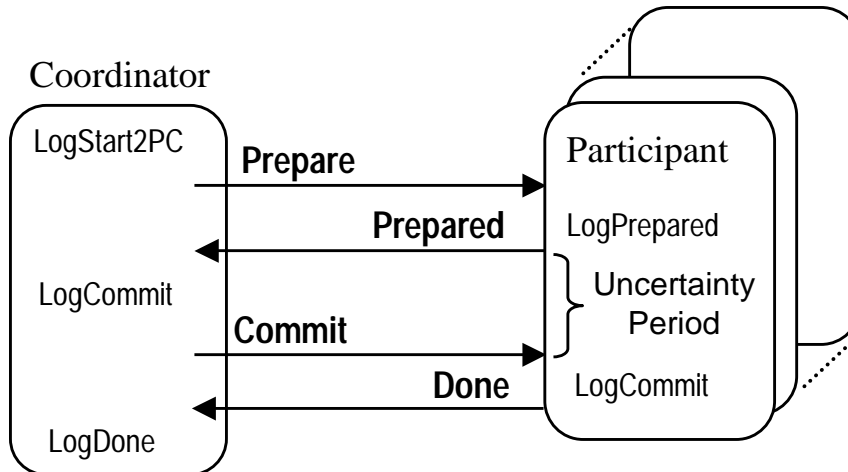


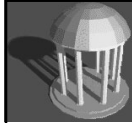


Two-Phase Commit



Two-Phase Commit





Failures Are Indicated By Timeout

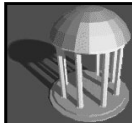
- ◆ What actions are taken if:

- A participant times out waiting for coordinator's PREPARE message?

- The coordinator times out waiting for a participant's response (PREPARED, NOT_PREPARED)?

- A participant that responded PREPARED times out waiting for the coordinator's decision?

- The coordinator times out waiting for DONE responses from participants?



Distributed Recovery Manager: Additional Restart Actions for T

- ◆ Coordinator

- If {T, COMMIT} not in log, send ABORT to participants

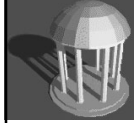
- If {T, COMMIT} is in log but {T, DONE} is not, send COMMIT to participants

- ◆ Participant:

- If {T, COMMIT} or {T, ABORT} in log, recover

- If {T, PREPARED} is *not* in log, Abort T

- If {T, PREPARED} is in log but {T, COMMIT} or {T, ABORT} is not, run the termination protocol



Some Fundamental Results

- ◆ No agreement (commit) protocol, including the two-phase commit protocol, can prevent blocking by some participant in the event of a failure until that failure is repaired.
- ◆ No commit protocol, including the two-phase commit protocol, can guarantee the independent recovery of a failed participant (the recovered participant can decide to commit or abort without communicating with other nodes).