
DISTRIBUTED SYSTEMS (COMP9243)

Lecture 6: Synchronisation and Coordination (Part 2)

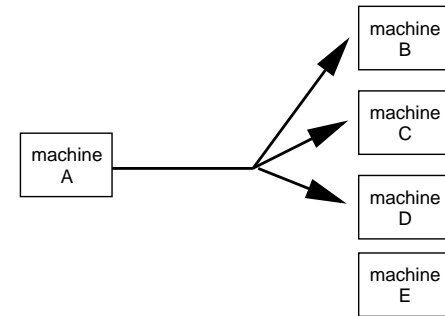
Slide 1

- ① Multicast
- ② Elections
- ③ Transactions

Slide 2

MULTICAST

Slide 3



- Sender performs a single `send()`
- Group of receivers
- Membership of group is transparent

Slide 4

EXAMPLES

Fault Tolerance:

- Replicated (redundant) servers
- Strong consistency: multicast operations

Service Discovery:

- Multicast request for service
- Reply from service provider

Performance:

- Replicated servers or data
- Weaker consistency: multicast operations or data

Event or Notification propagation:

- Group members are those interested in particular events
 - Example: sensor data, stock updates, network status
-

PROPERTIES

Group membership:

- Static: membership does not change
- Dynamic: membership changes

Open vs Closed group:

- Closed group: only members can send
- Open group: anyone can send

Slide 5

Reliability:

- Communication failure vs process failure
- Guarantee of delivery:
 - all members (or none) – Atomic
 - all non-failed members

Ordering:

- Guarantee of ordered delivery
 - FIFO, Causal, Total Order
-

EXAMPLES REVISITED

Fault Tolerance:

- Reliability: Atomic
- Ordering: Total
- Membership: Static
- Group: Closed

Service Discovery:

- Reliability: No guarantee
- Ordering: None
- Membership: Open
- Group: Static

Slide 6

Performance:

- Reliability: Non-failed
- Ordering: FIFO, Causal
- Membership: Closed
- Group: Dynamic

Event or Notification propagation:

- Reliability: Non-failed
 - Ordering: Causal
 - Membership: Open
 - Group: Dynamic
-

OTHER ISSUES

Performance:

- Bandwidth
- Delay

Efficiency:

- Avoid sending a message over a link multiple times (stress)
- Distribution tree
- Hardware support (e.g., Ethernet broadcast)

Slide 7

Network-level vs Application-level:

- Network routers understand multicast
 - Applications (or middleware) send unicasts to group members
 - Overlay distribution tree
-

NETWORK-LEVEL MULTICAST

"You put packets in at one end, and the network conspires to deliver them to anyone who asks." Dave Clark

Ethernet Broadcast:

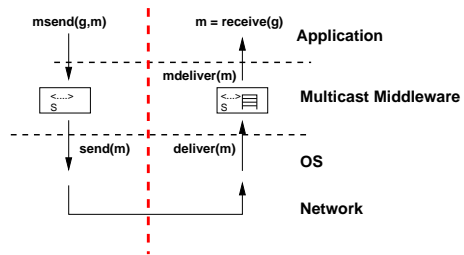
- all hosts on local network
- MAC address: FF:FF:FF:FF:FF:FF

Slide 8

IP Multicast:

- multicast group: class D Internet address:
 - first 4 bits: 1110 (224.0.0.0 to 239.255.255.255)
 - permanent groups: 224.0.0.1 - 224.0.0.255
 - multicast routers
 - join group: Internet Group Management Protocol (IGMP)
 - set distribution trees: Protocol Independent Multicast (PIM)
-

APPLICATION-LEVEL MULTICAST SYSTEM MODEL



Slide 9

Assumptions:

- reliable one-to-one channels
- no failures
- single closed group

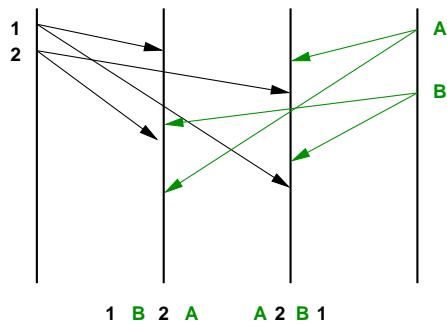
```

B-send(g,m) {
  foreach p in g {
    send(p, m);
  }
}

deliver(m) {
  B-deliver(m);
}
    
```

Slide 11

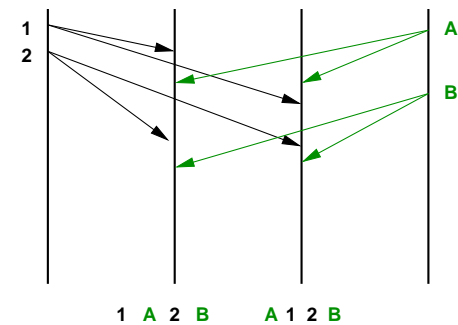
BASIC MULTICAST



Slide 10

- no reliability guarantees
- no ordering guarantees

FIFO MULTICAST



Slide 12

- order maintained per sender

Slide 13

```

FO-init() {
  S = 0;           // local sequence #
  for (i = 1 to N) V[i] = 0; // vector of last seen seq #s
}

FO-send(g, m) {
  S++;
  B-send(g, <m,S>); // multicast to everyone
}

```

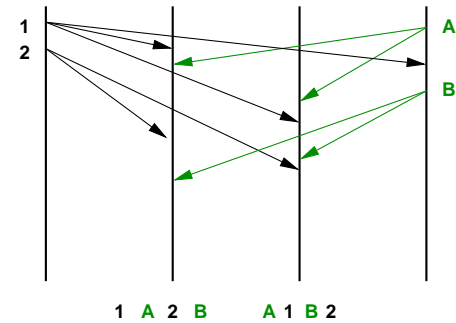
Slide 14

```

B-deliver(<m,S>) {
  if (S == V[sender(m)] + 1) {
    // expecting this msg, so deliver
    FO-deliver(m);
    V[sender(m)] = S;
  } else if (S > V[sender(m)] + 1) {
    // not expecting this msg, so put in queue for later
    enqueue(<m,S>);
  }
  // check if msgs in queue have become deliverable
  foreach <m,S> in queue {
    if (S == V[sender(m)] + 1) {
      FO-deliver(m);
      dequeue(<m,S>);
      V[sender(m)] = S;
    } } }
}

```

CAUSAL MULTICAST



Slide 15

- order maintained between causally related sends
- 1 and A, 2 and B are concurrent
- 1 happens before B

Slide 16

```

CO-init() {
  // vector of what we've delivered already
  for (i = 1 to N) V[i] = 0;
}

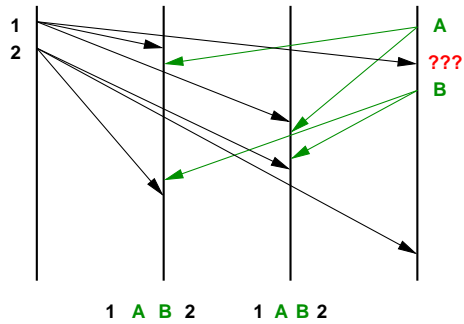
CO-send(g, m) {
  V[i]++;
  B-send(g, <m,V>);
}

B-deliver(<m,Vj>) { // j = sender(m)
  enqueue(<m,Vj>);
  // make sure we've delivered everything the message
  // could depend on
  wait until Vj[j] == V[j] + 1 and Vj[k] <= V[k] (k!= j)
  CO-deliver(m);
  dequeue(<m,Vj>); V[j]++;
}

```

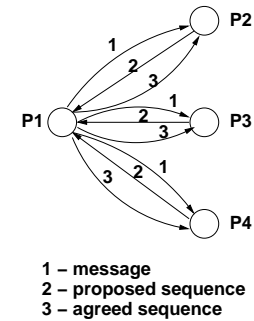
Slide 17

TOTALLY ORDERED MULTICAST



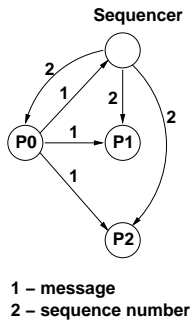
Agreement-based:

Slide 19



Slide 18

Sequencer Based:



Slide 20

Other possibilities:

- Moving sequencer
- Logical clock based
 - each receiver determines order independently
 - delivery based on sender timestamp ordering
 - how do you know you have most recent timestamp?
- Token based
- Physical clock ordering

Hybrid Ordering:

- FIFO + Total
- Causal + Total

Dealing with Failure:

- Communication
- Process

Slide 21

ELECTIONS

Determining a coordinator:

- Some algorithms rely on a distinguished coordinator process
- Coordinator needs to be determined
- May also need to change coordinator at runtime
- Assume all nodes have unique id
- Election: agree on which non-crashed process has largest id number

Slide 22

Requirements:

- ① **Safety:** A process either doesn't know the coordinator or it knows the id of the process with largest id number
- ② **Liveness:** Eventually, a process crashes or knows the coordinator

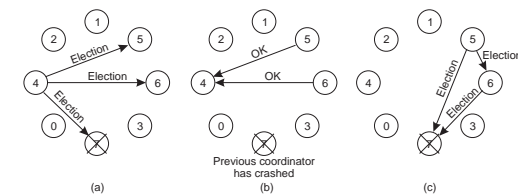
BULLY ALGORITHM

→ Three types of messages:

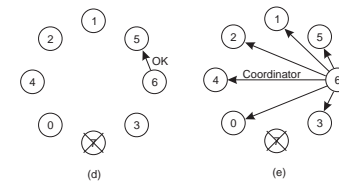
- *Election*: announce election
- *Answer*: response to election
- *Coordinator*: announce elected coordinator

- A process begins an election when it notices through a timeout that the coordinator has failed or receives an *Election* message
- When starting an election, send *Election* to all higher-numbered processes
- If no *Answer* is received, the election starting process is the coordinator and sends a *Coordinator* message to all other processes
- If an *Answer* arrives, it waits a predetermined period of time for a *Coordinator* message
- If a process knows it is the highest numbered one, it can immediately answer with *Coordinator*

Slide 23

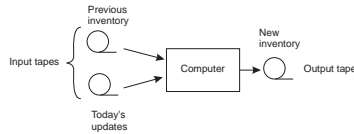


Slide 24



TRANSACTION EXAMPLES

Inventory:



Slide 29

Banking:

```
BeginTransaction
  b = A.Balance();
  A.Withdraw(b);
  B.Deposit(b);
EndTransaction
```

Slide 30

ACID PROPERTIES

atomic: all-or-nothing. once committed the full transaction is performed, if aborted, there is no trace left;

consistent: concurrent transactions will not produce inconsistent results;

isolated: transactions do not interfere with each other i.e. no intermediate state of a transaction is visible outside (also called serialisable);

durable: after a commit, results are permanent (even if server or hardware fails)

CLASSIFICATION OF TRANSACTIONS

Flat: sequence of operations that satisfies ACID

Nested: *hierarchy* of transactions

Distributed: (flat) transaction that is executed on distributed data

Slide 31

Flat Transactions:

- ✓ Simple
- ✗ Failure → all changes undone

```
BeginTransaction
  accountA -= 100;
  accountB += 50;
  accountC += 25;
  accountD += 25;
EndTransaction
```

Slide 32

NESTED TRANSACTION

Example:

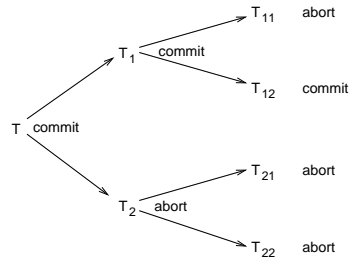
Booking a flight

- ✓ Sydney → Manila
- ✓ Manila → Amsterdam
- ✗ Amsterdam → Toronto

What to do?

- Abort whole transaction
- Partially commit transaction and try alternative for aborted part
- Commit nonaborted parts of transaction

Slide 33



- Subtransactions and parent transactions
- Parent transaction may commit even if some subtransactions aborted
- Parent transaction aborts → all subtransactions abort

Slide 34

Subtransactions:

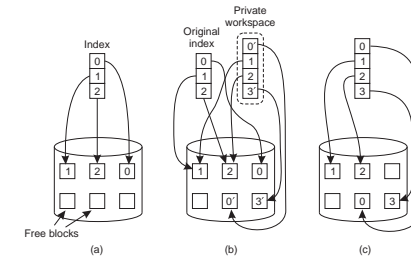
- Subtransaction can abort any time
- Subtransaction cannot commit until parent ready to commit
 - Subtransaction either aborts or commits provisionally
 - Provisionally committed subtransaction reports provisional commit list, containing all its provisionally committed subtransactions, to parent
 - On abort, all subtransactions in that list are aborted.

TRANSACTION IMPLEMENTATION

Private Workspace:

- Perform all tentative operations on a shadow copy
- Atomically swap with main copy on Commit
- Discard shadow on Abort.

Slide 35



Writeahead Log:

- In-place update with writeahead logging
- Roll back on Abort

Slide 36

<code>x = 0;</code>			
<code>y = 0;</code>			
<code>BEGIN_TRANSACTION;</code>	Log	Log	Log
<code> x = x + 1;</code>	[x = 0/1]	[x = 0/1]	[x = 0/1]
<code> y = y + 2;</code>		[y = 0/2]	[y = 0/2]
<code> x = y * y;</code>			[x = 1/4]
<code>END_TRANSACTION;</code>			
(a)	(b)	(c)	(d)

CONCURRENCY CONTROL

Simultaneous Transactions:

- Clients accessing bank accounts
- Travel agents booking flights
- Inventory system updated by cash registers

Problems:

- Slide 37**
- Simultaneous transactions may interfere
 - Lost update
 - Inconsistent retrieval
 - Consistency and Isolation require that there is no interference

Concurrency Control Algorithms:

- Guarantee that multiple transactions can be executed simultaneously while still being isolated.
- As though transactions executed one after another

CONFLICTS AND SERIALISABILITY

Read/Write Conflicts Revisited:

conflict: operations (from the same, or different transactions) that operate on same data

Slide 38 **read-write conflict:** one of the operations is a write

write-write conflict: more than one operation is a write

Schedule:

- Total ordering (interleaving) of operations
- Legal schedules provide results as though transactions serialised (*serial equivalence*)

Example Schedules:

```

BEGIN_TRANSACTION  BEGIN_TRANSACTION  BEGIN_TRANSACTION
  x = 0;             x = 0;             x = 0;
  x = x + 1;         x = x + 2;         x = x + 3;
END_TRANSACTION    END_TRANSACTION    END_TRANSACTION
    
```

Slide 39

	(a)	(b)	(c)	
	Time →			
Schedule 1	x = 0; x = x + 1;	x = 0; x = x + 2;	x = 0; x = x + 3;	Legal
Schedule 2	x = 0; x = 0;	x = x + 1; x = x + 2;	x = 0; x = x + 3;	Legal
Schedule 3	x = 0; x = 0;	x = x + 1; x = 0;	x = x + 2; x = x + 3;	Illegal

(d)

SERIALISABLE EXECUTION

Serial Equivalence:

- conflicting operations performed in same order on all data items
 - operation in T_1 before T_2 , or
 - operation in T_2 before T_1

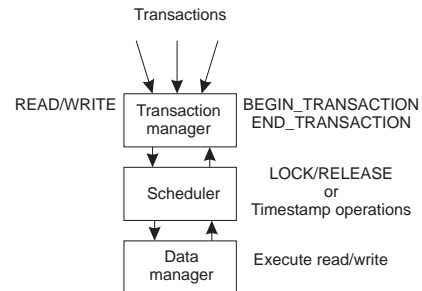
Slide 40

Are the following serially equivalent?

- $R_1(x)W_1(x)R_2(y)W_2(y)R_2(x)W_1(y)$
- $R_1(x)R_2(y)W_2(y)R_2(x)W_1(x)W_1(y)$
- $R_1(x)R_2(x)W_1(x)W_2(y)R_2(y)W_1(y)$
- $R_1(x)W_1(x)R_2(x)W_2(y)R_2(y)W_1(y)$

MANAGING CONCURRENCY

Transaction Managers:



Slide 41

LOCKING

Pessimistic approach: prevent illegal schedules

Slide 43

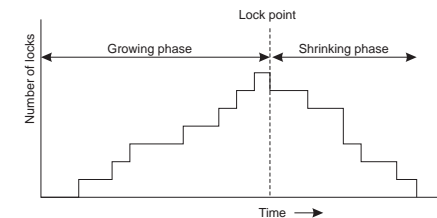
- Lock must be obtained from scheduler before a read or write.
- Scheduler grants and releases locks
- Ensures that only valid schedules result

Dealing with Concurrency:

Slide 42

- Locking
- Timestamp Ordering
- Optimistic Control

TWO PHASE LOCKING (2PL)



Slide 44

- ① Lock granted if no conflicting locks on that data item. Otherwise operation delayed until lock released.
- ② Lock is not released until operation executed by data manager
- ③ No more locks granted after a release has taken place

All schedules formed using 2PL are serialisable.

PROBLEMS WITH LOCKING

Deadlock:

- Detect and break deadlocks (in scheduler)
- Timeout on locks

Slide 45

Cascaded Aborts:

- $Release(T_i, x) \rightarrow Lock(T_j, x) \rightarrow Abort(T_i)$
- T_j will have to be aborted too

solution: Strict Two-Phase Locking:

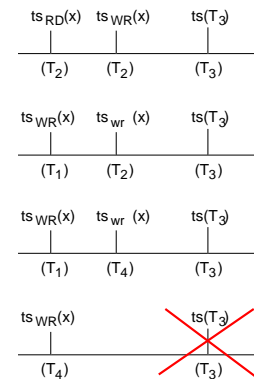
- Release *all* locks at Commit/Abort

TIMESTAMP ORDERING

- Each transaction has unique timestamp ($ts(T_i)$)
- Each operation ($TS(W), TS(R)$) receives its transaction's timestamp
- Each data item has two timestamps:
 - read timestamp: $ts_{RD}(x)$ - transaction that most recently read x
 - write timestamp: $ts_{WR}(x)$ - committed transaction that most recently wrote x
- Also tentative write timestamps (noncommitted writes) $ts_{wr}(x)$
- Timestamp ordering rule:
 - write request only valid if $TS(W) > ts_{WR}$ and $TS(W) \geq ts_{RD}$
 - read request only valid if $TS(R) > ts_{WR}$
- Conflict resolution:
 - Operation with lower timestamp executed first

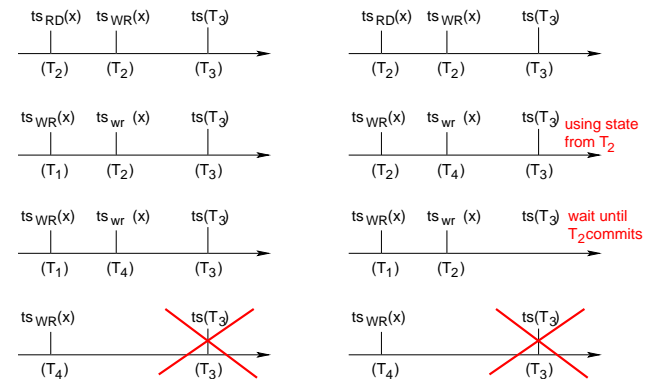
Slide 46

Write



Slide 47

Read



OPTIMISTIC CONTROL

Assume that no conflicts will occur.

- Detect conflicts at commit time
- Three phases:
 - Working (using shadow copies)
 - Validation
 - Update

Slide 48

Slide 49

Validation:

- Keep track of read set and write set during working phase
- During validation make sure conflicting operations with *overlapping* transactions are serialisable
 - Make sure T_v doesn't read items written by other T_i s
 - Make sure T_v doesn't write items read by other T_i s
 - Make sure T_v doesn't write items written by other T_i s
- Prevent overlapping of validation phases (mutual exclusion)

Slide 50

Backward validation:

- Check committed overlapping transactions
- Only have to check if T_v read something another T_i has written
- Abort T_v if conflict
 - ✗ Have to keep old write sets

Forward validation:

- Check not yet committed overlapping transactions
- Only have to check if T_v wrote something another T_i has read
- Options on conflict: abort T_v , abort T_i , wait
 - ✗ Read sets of not yet committed transactions may change during validation!

DISTRIBUTED TRANSACTIONS

Slide 51

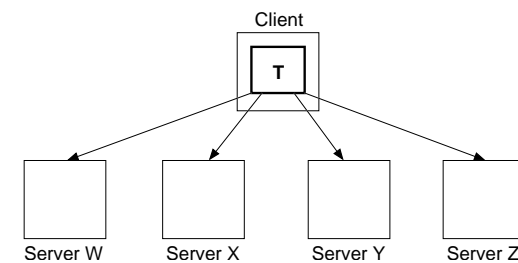
- In distributed system, a single transaction will, in general, involve several servers:
 - transaction may require several services,
 - transaction involves files stored on different servers
- All servers must agree to *Commit* or *Abort*, and do this atomically.

Transaction Management:

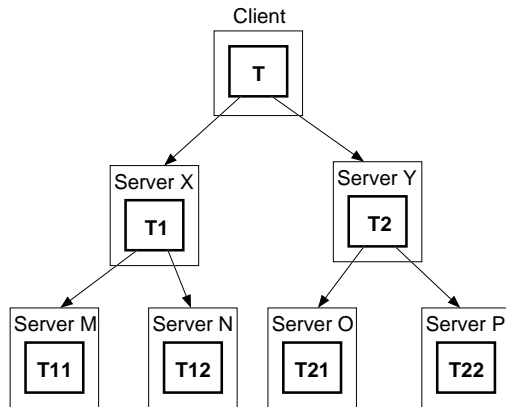
- Centralised
- Distributed

Slide 52

Distributed Flat Transaction:



Distributed Nested Transaction:



Slide 53

DISTRIBUTED LOCKING

Centralised 2PL:

- Single server handles all locks
- Scheduler only grants locks, transaction manager contacts data manager for operation.

Primary 2PL:

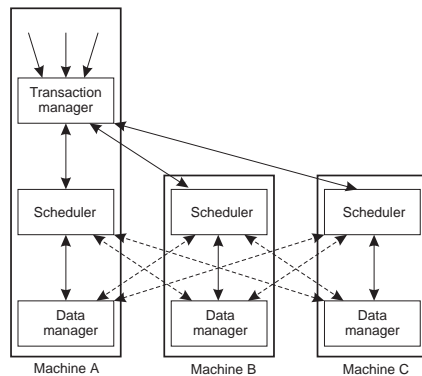
- Each data item is assigned a primary copy
- Scheduler on that server responsible for locks

Distributed 2PL:

- Data can be replicated
- Scheduler on each machine responsible for locking own data
- Read lock: contact any replica
- Write lock: contact all replicas

Slide 55

DISTRIBUTED CONCURRENCY CONTROL



Slide 54

Distributed Timestamps:

Assigning unique timestamps:

- Timestamp assigned by first scheduler accessed
- Clocks have to be roughly synchronized

Slide 56

Distributed Optimistic Control:

- Validation operations distributed over servers
- Commitment deadlock (because of mutual exclusion of validation)
- Parallel validation protocol
- Make sure that transaction serialised correctly

ATOMICITY AND DISTRIBUTED TRANSACTIONS

Distributed Transaction Organisation:

Slide 57

- Each distributed transaction has a **coordinator**, the server handling the initial `BeginTransaction` call
- Coordinator maintains a list of **workers**, i.e. other servers involved in the transaction
- Each worker needs to know coordinator
- Coordinator is responsible for ensuring that whole transaction is atomically committed or aborted
 - Require a **distributed commit protocol**.

DISTRIBUTED ATOMIC COMMIT

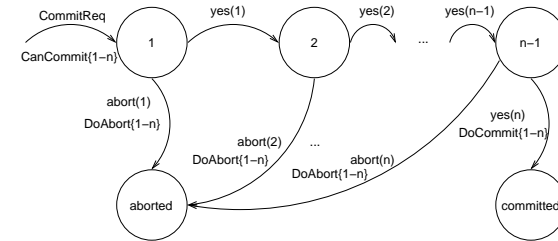
Slide 58

- Transaction may only be able to commit when all workers are ready to commit (e.g. validation in optimistic concurrency)
- Hence distributed commit requires at least two phases:
 1. **Voting phase**: all workers vote on commit, coordinator then decides whether to commit or abort.
 2. **Completion phase**: all workers commit or abort according to decision.

Basic protocol is called **two-phase commit** (2PC)

Two-phase commit: Coordinator:

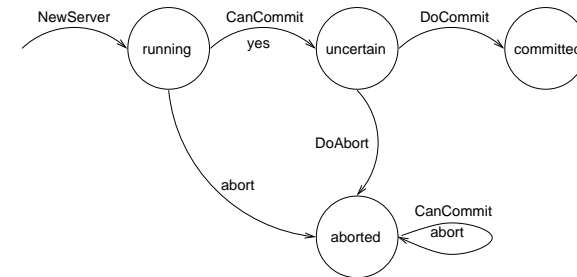
Slide 59



1. sends `CanCommit`, receives `yes`, `abort`;
2. sends `DoCommit`, `DoAbort`

Two-phase commit: Worker:

Slide 60



1. receives `CanCommit`, sends `yes`, `abort`;
2. receives `DoCommit`, `DoAbort`

Failures can be due to:

→ **server failures:**

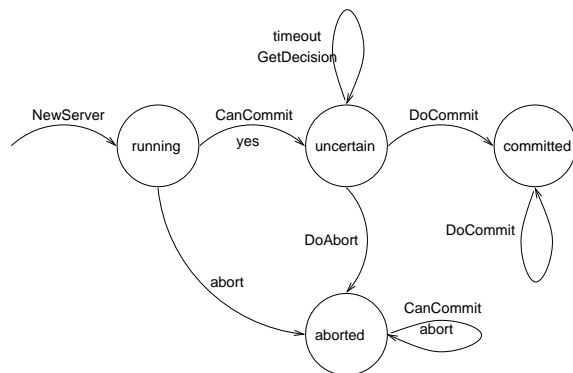
- restarting worker aborts all transactions.

Slide 61

→ **Failure of communication channels:**

- coordinator aborts after timeout.

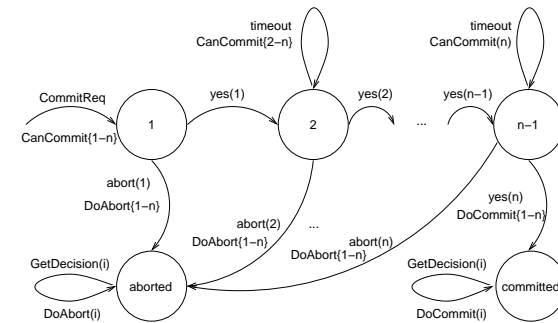
Two-phase commit with timeouts: Worker:



Slide 62

→ On *timeout* sends *GetDecision*.

Two-phase commit with timeouts: Coordinator:



Slide 63

→ On *timeout* re-sends *CanCommit*, On *GetDecision* repeats decision.

Limitations:

→ Once node voted "yes", cannot change its mind, even if crashes.

Slide 64

→ If coordinator crashes, all workers may be blocked.

- Atomic state update to ensure "yes" vote is stable.
- Can use different protocols (e.g. three-phase commit),
- in some circumstances workers can obtain result from other workers.

Slide 65

Two-phase commit of nested transactions:

- Two-phase commit is required, as a worker might crash after provisional commit
- On `CanCommit` request, worker:
 - votes "no": if it has no recollection of subtransactions of committing transaction (i.e. must have crashed recently),
 - otherwise
 - aborts subtransactions of aborted transactions,
 - saves provisionally committed transactions in stable store,
 - votes "yes".

Two Approaches:

- Hierarchic 2PC
 - Flat 2PC
-