Slide 1

DISTRIBUTED SYSTEMS (COMP9243)

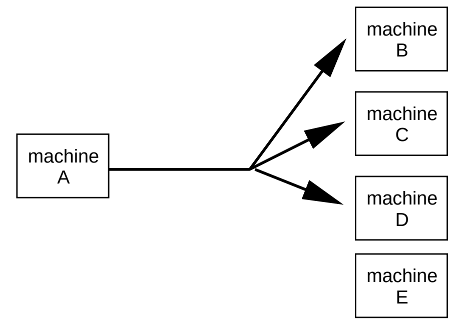
Lecture 6: Synchronisation and Coordination (Part 2)

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

Slide 5

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

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

Slide 6

Examples Revisited

Fault Tolerance:

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

Service Discovery:

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

Performance:

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

Event or Notification propagation:

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

Slide 7

Other Issues

Performance:

- ➜ Bandwidth
- ➜ Delay

Efficiency:

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

Network-level vs Application-level:

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

Slide 8

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:

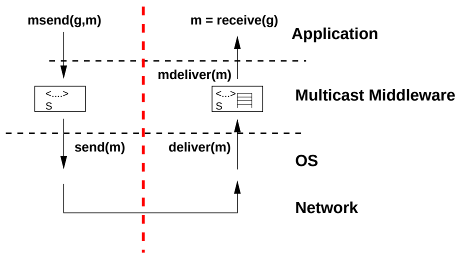
- ➜ all hosts on local network
- ➜ MAC address: FF:FF:FF:FF:FF:FF

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)

Slide 9

Application-level Multicast System Model

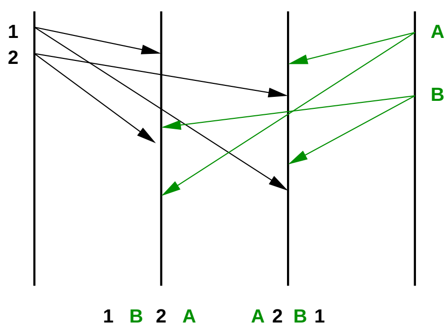


Assumptions:

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

Slide 10

Basic Multicast



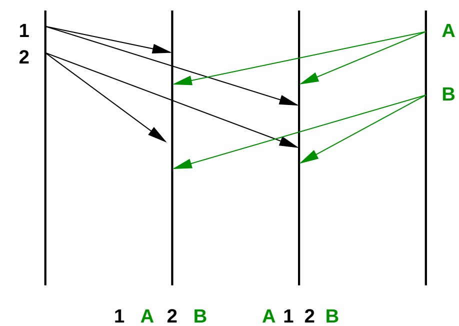
- ➜ no reliability guarantees
- ➜ no ordering guarantees

Slide 11

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

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

Slide 12

FIFO Multicast



- ➜ order maintained per sender

Slide 13

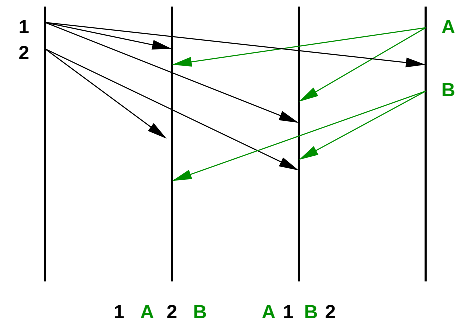
```
FO-init() {
  S = 0;              // local sequence #
  for (i = 1 to N) V[i] = 0; // vector of last seen seq #s
}

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;
} } }
```

Slide 15

CAUSAL MULTICAST



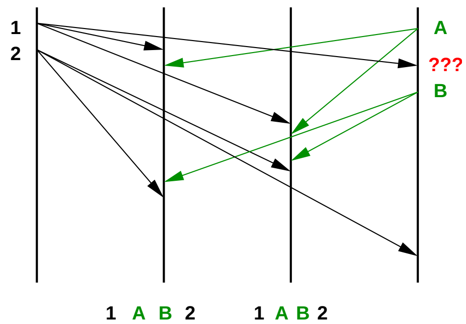
- ➜ order maintained between causally related sends
- ➜ 1 and A, 2 and B are concurrent
- ➜ 1 happens before B

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]++;
}
```

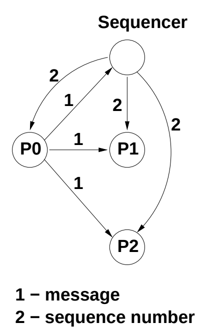
Slide 17

Totally Ordered Multicast



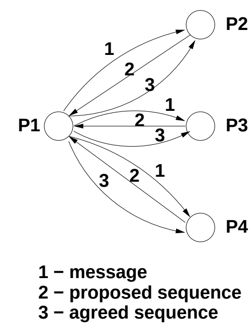
Slide 18

Sequencer Based:



Slide 19

Agreement-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

Slide 22

Coordinator:

- ➜ Some algorithms rely on a distinguished coordinator process
- ➜ Coordinator needs to be determined
- ➜ May also need to change coordinator at runtime

Election:

- ➜ Goal: when algorithm finished all processes agree who new coordinator is.

Slide 23

Determining a coordinator:

- ➜ Assume all nodes have unique id
- ➜ possible assumption: processes know all other process's ids but don't know if they are up or down
- ➜ Election: agree on which non-crashed process has largest id number

Requirements:

① **Safety:** A process either doesn't know the coordinator or it knows the id of the process with largest id number

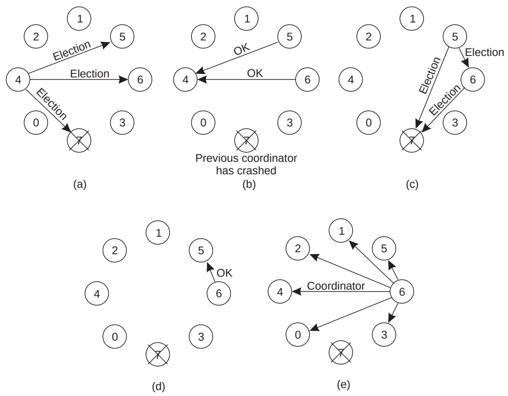
② **Liveness:** Eventually, a process crashes or knows the coordinator

Slide 24

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 25

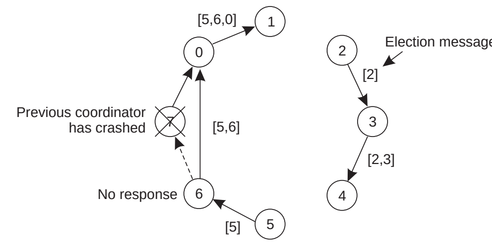


Slide 26

Ring Algorithm

- ➜ Two types of messages:
  - *Election:* forward election data
  - *Coordinator:* announce elected coordinator
- ➜ Processes ordered in ring
- ➜ A process begins an election when it notices through a timeout that the coordinator has failed.
- ➜ Sends message to first neighbour that is up
- ➜ Every node adds own id to *Election* message and forwards along the ring
- ➜ Election finished when originator receives *Election* message again
- ➜ Forwards message on as *Coordinator* message

Slide 27



Slide 28

Transactions

Slide 29

TRANSACTIONS

Transaction:

- ➜ Comes from database world
- ➜ Defines a sequence of operations
- ➜ Atomic in presence of multiple clients and failures

Mutual Exclusion ++:

- ➜ Protect shared data against simultaneous access
- ➜ Allow multiple data items to be modified in single atomic action

Transaction Model:

Operations:

- ➜ `BeginTransaction`
- ➜ `EndTransaction`
- ➜ `Read`
- ➜ `Write`

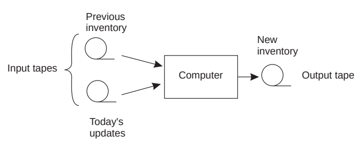
End of Transaction:

- ➜ Commit
- ➜ Abort

Slide 30

TRANSACTION EXAMPLES

Inventory:



Banking:

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

Slide 31

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)

Slide 32

CLASSIFICATION OF TRANSACTIONS

Flat: sequence of operations that satisfies ACID

Nested: *hierarchy* of transactions

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

Flat Transactions:

- ☑ Simple
- ✗ Failure → all changes undone

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

Slide 33

Nested Transaction

Example:

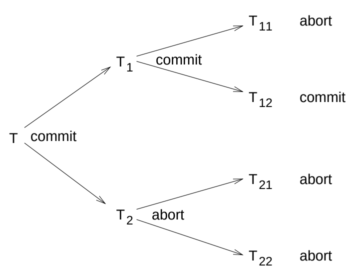
Booking a flight

- ✓ 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 34



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

Slide 35

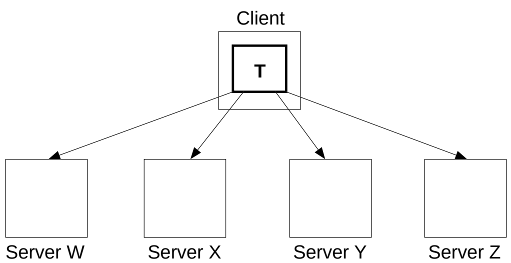
Subtransactions:

- ➜ 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.

Slide 36

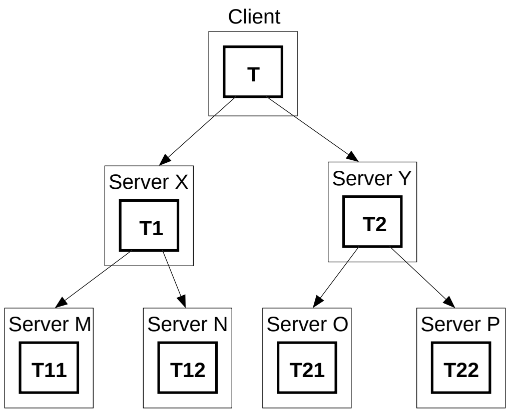
Distributed Transaction

Distributed Flat Transaction:



Slide 37

Distributed Nested Transaction:

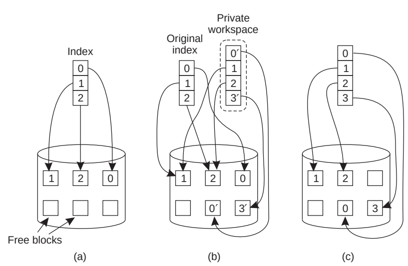


Slide 38

TRANSACTION ATOMICITY IMPLEMENTATION

Private Workspace:

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



Slide 39

Writeahead Log:

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

```
x = 0;
y = 0;
BEGIN_TRANSACTION;
   x = x + 1;
   y = y + 2;
   x = y * y;
END_TRANSACTION;
```

(a)

Log (b)	Log (c)	Log (d)
[x = 0/1]	[x = 0/1]	[x = 0/1]
	[y = 0/2]	[y = 0/2]
		[x = 1/4]

Slide 40

CONCURRENCY CONTROL

Simultaneous Transactions:

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

Problems:

- ➜ 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

Slide 41

Conflicts and Serialisability

Read/Write Conflicts Revisited:

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

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*)

Slide 42

Example Schedules:

```
BEGIN_TRANSACTION
   x = 0;
   x = x + 1;
END_TRANSACTION
```
(a)

```
BEGIN_TRANSACTION
   x = 0;
   x = x + 2;
END_TRANSACTION
```
(b)

```
BEGIN_TRANSACTION
   x = 0;
   x = x + 3;
END_TRANSACTION
```
(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)

Slide 43

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

Are the following serially equivalent?

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

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

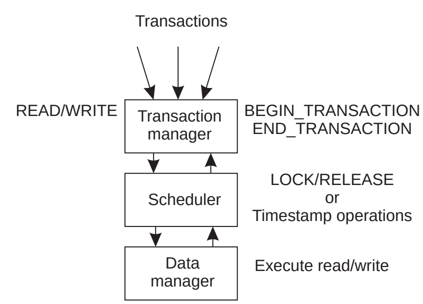
➜ $R_1(x)R_2(x)W_1(x)W_2(y)R_2(y)W_1(y)$

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

Slide 44

Managing Concurrency

Transaction Managers:



Slide 45

Dealing with Concurrency:

- ➜ Locking
- ➜ Timestamp Ordering
- ➜ Optimistic Control

Slide 46

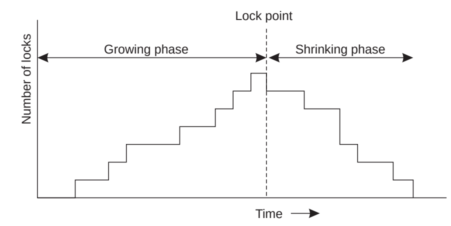
Locking

Pessimistic approach: prevent illegal schedules

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

Slide 47

Two Phase Locking (2PL)



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.

Slide 48

Problems with Locking

Deadlock:

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

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:

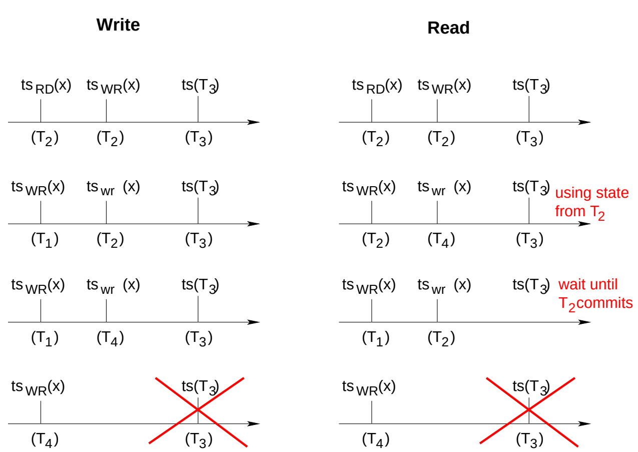
- ➜ Release *all* locks at `Commit/Abort`

Timestamp Ordering

Slide 49

- 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 50



Optimistic Control

Slide 51

Assume that no conflicts will occur.

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

Slide 52

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_is
  - Make sure T_v doesn't write items read by other T_is
  - Make sure T_v doesn't write items written by other T_is
- Prevent overlapping of validation phases (mutual exclusion)

Slide 53

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!

Slide 54

Distributed Transactions

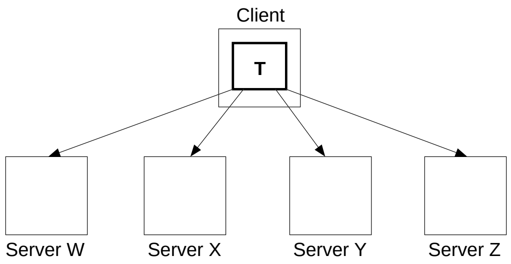
- ➜ In distributed system, a single transaction will, in general, involve several servers:
  - transaction may require several services,
  - transaction involves files stored on different servers
- ➜ All servers must agree to *Commit* or *Abort*, and do this atomically.

Transaction Management:

- ➜ Centralised
- ➜ Distributed

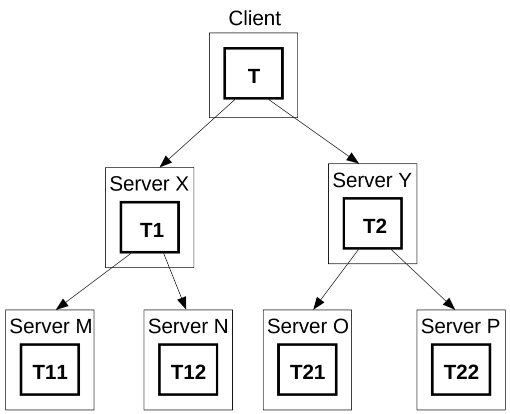
Slide 55

Distributed Flat Transaction:



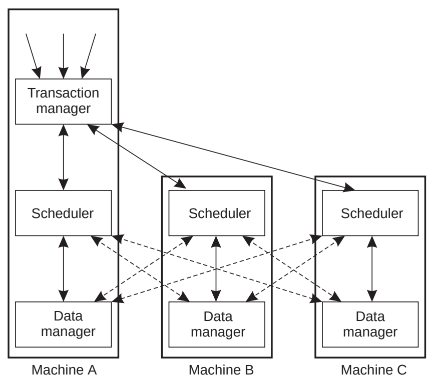
Slide 56

Distributed Nested Transaction:



Slide 57

Distributed Concurrency Control



Slide 58

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 59

Distributed Timestamps:

Assigning unique timestamps:

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

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

Slide 60

Atomicity and Distributed Transactions

Distributed Transaction Organisation:

- ➜ 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.

Slide 61

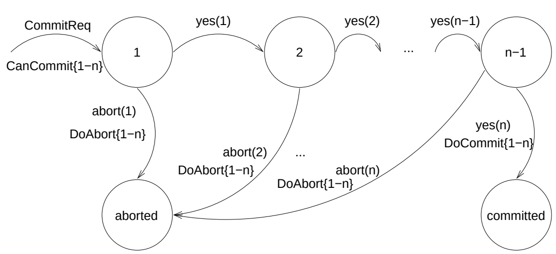
DISTRIBUTED ATOMIC COMMIT

- ➜ 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)

Slide 62

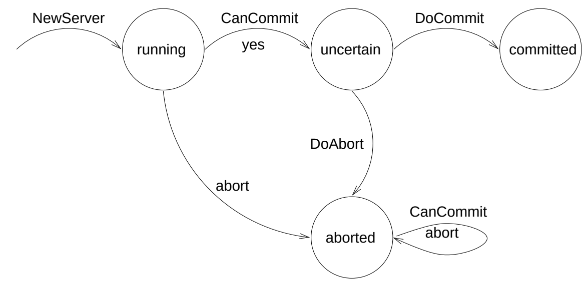
Two-phase commit: Coordinator:



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

Slide 63

Two-phase commit: Worker:



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

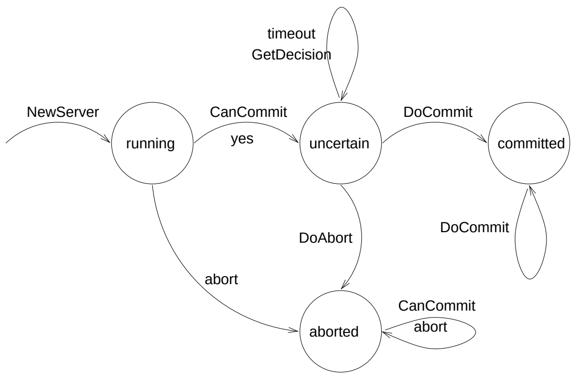
Slide 64

Failures can be due to:

- ➜ **server failures:**
  - restarting worker aborts all transactions.
- ➜ **Failure of communication channels:**
  - coordinator aborts after timeout.

Slide 65

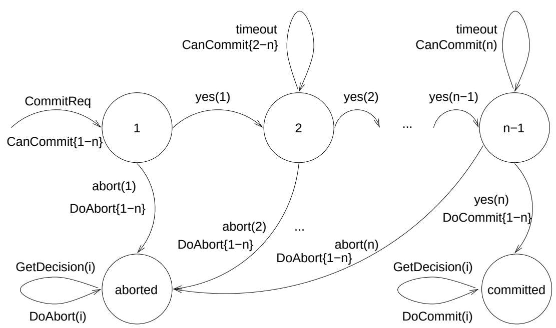
Two-phase commit with timeouts: Worker:



➜ On *timeout* sends `GetDecision`.

Slide 66

Two-phase commit with timeouts: Coordinator:



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

Slide 67

Limitations:

➜ Once node voted "yes", cannot change its mind, even if crashes.
- Atomic state update to ensure "yes" vote is stable.

➜ If coordinator crashes, all workers may be blocked.
- Can use different protocols (e.g. three-phase commit),
- in some circumstances workers can obtain result from other workers.

Slide 68

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