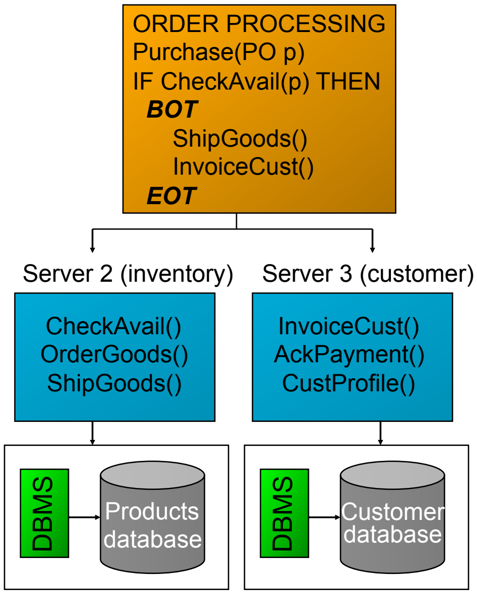
Distributed Transactions

Dominic Duggan

Stevens Institute of Technology

Based in part on materials by K. Birman

Client, server, and databases



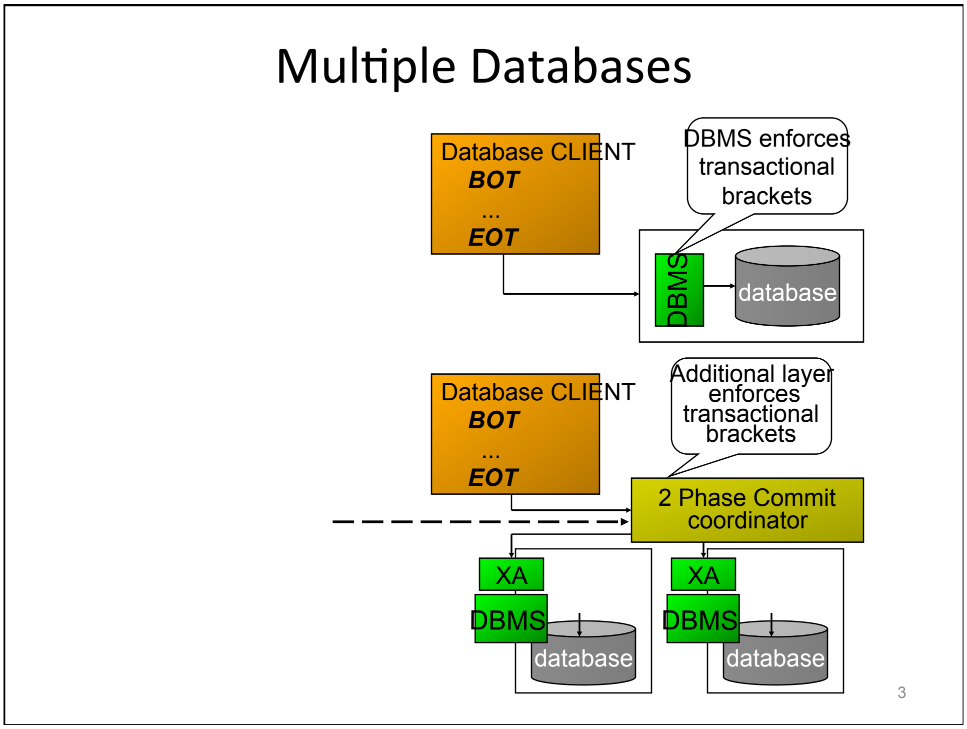
```
ORDER PROCESSING
Purchase(PO p)
IF CheckAvail(p) THEN
  BOT
    ShipGoods()
    InvoiceCust()
  EOT
```

Server 2 (inventory)

CheckAvail()
OrderGoods()
ShipGoods()

Server 3 (customer)

InvoiceCust()
AckPayment()
CustProfile()



Multiple Databases

Database CLIENT
BOT
...
EOT

DBMS enforces transactional brackets

DBMS

database

Database CLIENT
BOT
...
EOT

Additional layer enforces transactional brackets

2 Phase Commit coordinator

XA

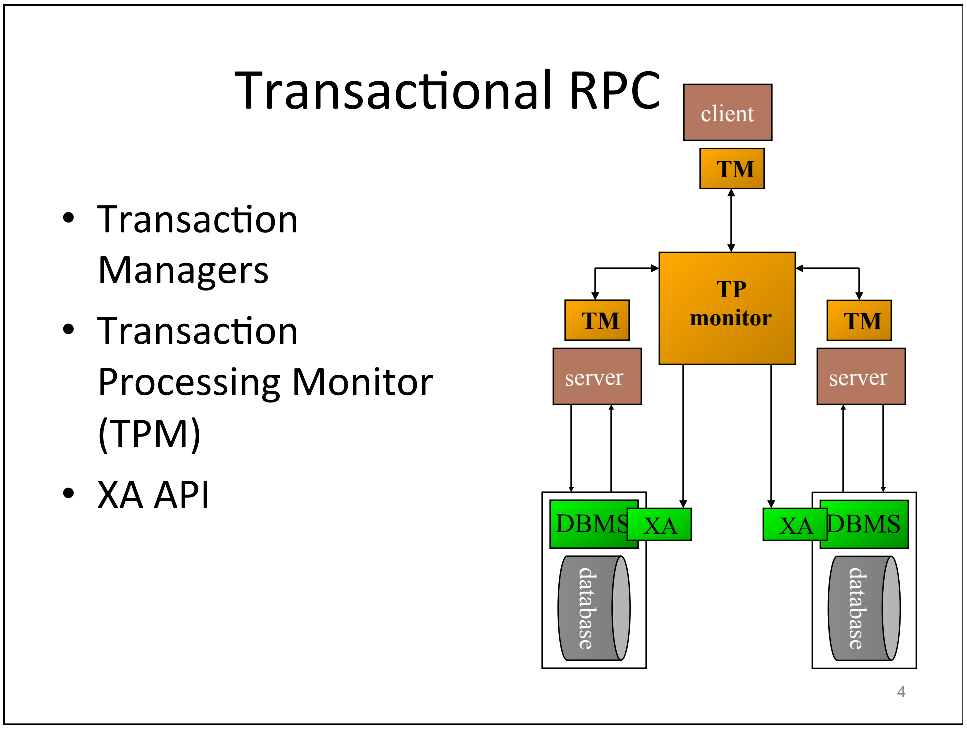
DBMS

database

XA

DBMS

database



Transactional RPC

- Transaction Managers
- Transaction Processing Monitor (TPM)
- XA API

client

TM

TP monitor

TM

server

TM

server

DBMS

XA

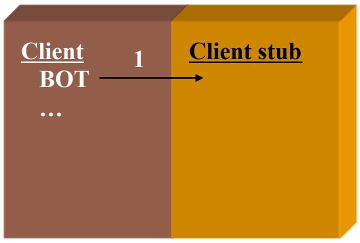
XA

DBMS

database

database

Basic TRPC (making calls)

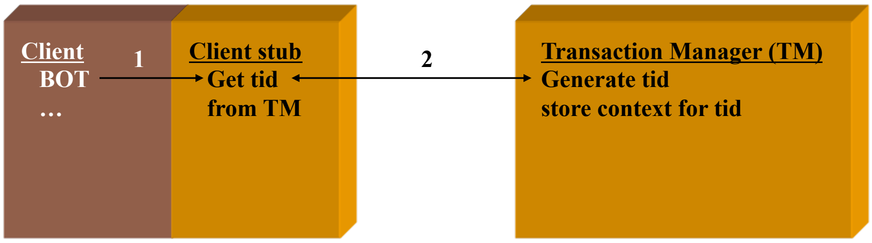


Client
BOT
…

1

Client stub

Basic TRPC (making calls)



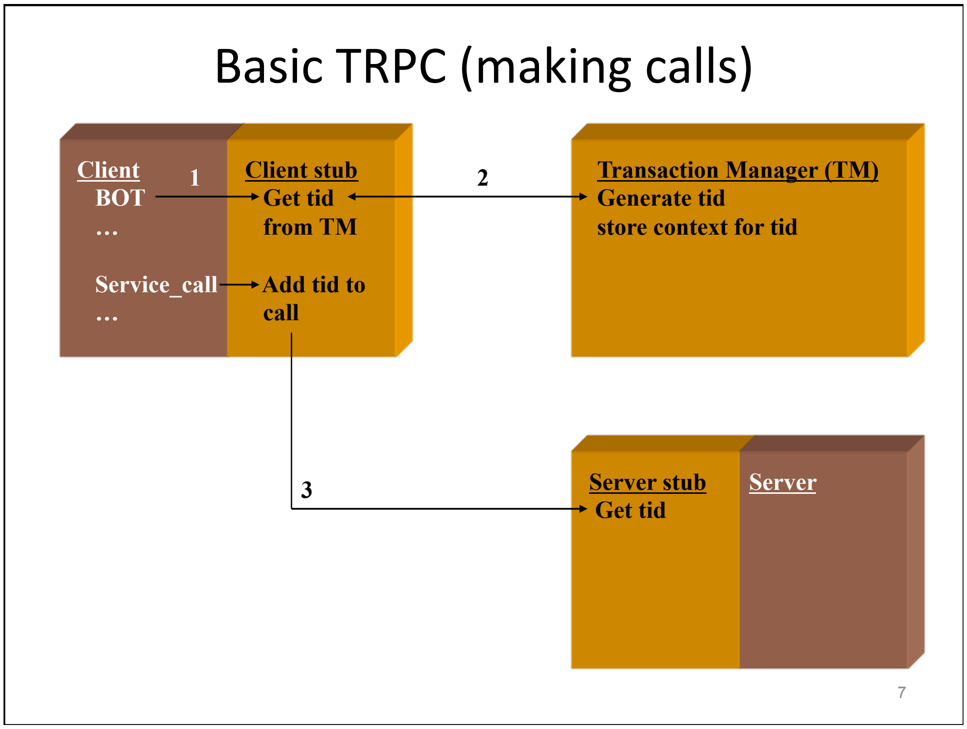
Client
BOT
…

1

Client stub
Get tid
from TM

2

Transaction Manager (TM)
Generate tid
store context for tid



Basic TRPC (making calls)

Client
BOT
…
Service_call
…

1

Client stub
Get tid
from TM
Add tid to
call

2

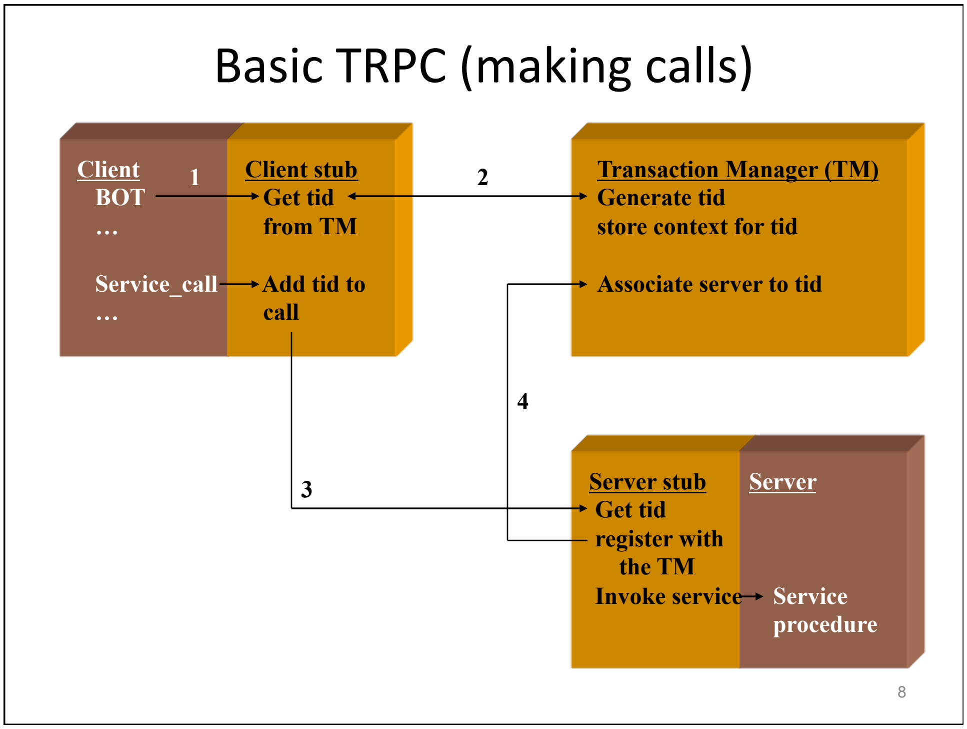
Transaction Manager (TM)
Generate tid
store context for tid

3

Server stub
Get tid

Server

7



Basic TRPC (making calls)

Client
BOT
…
Service_call
…

1

Client stub
Get tid
from TM
Add tid to
call

2

Transaction Manager (TM)
Generate tid
store context for tid

Associate server to tid

4

3

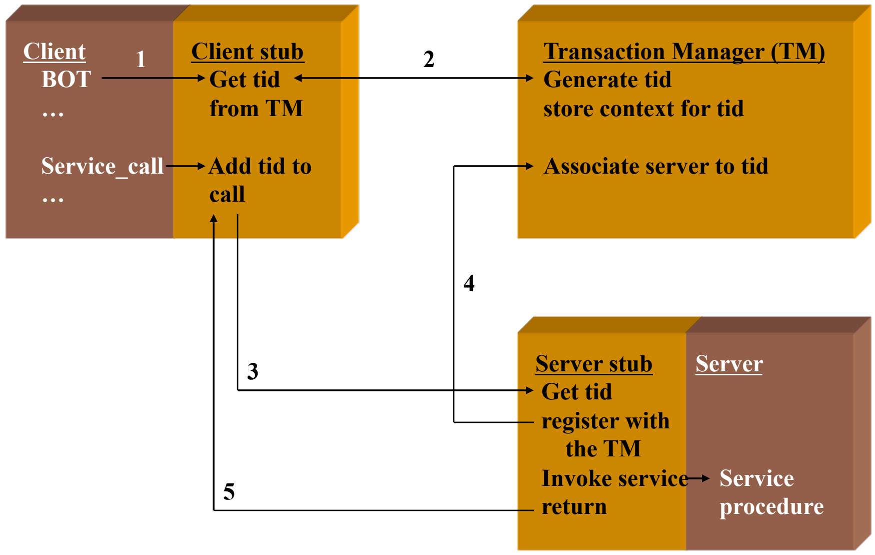
Server stub
Get tid
register with
the TM
Invoke service

Server

Service
procedure

8

Basic TRPC (making calls)



Client
BOT
…
Service_call
…

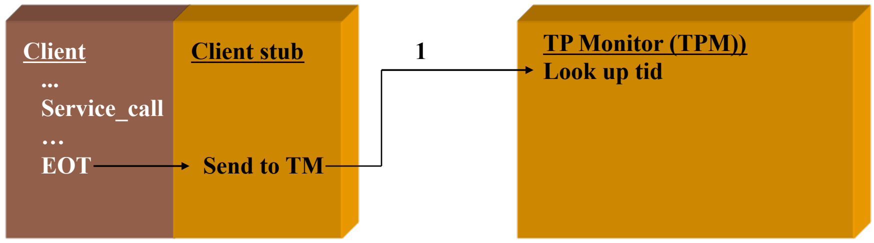
Client stub
Get tid from TM
Add tid to call

Transaction Manager (TM)
Generate tid
store context for tid
Associate server to tid

Server stub
Get tid
register with the TM
Invoke service
return

Server
Service procedure

Basic TRPC (committing calls)

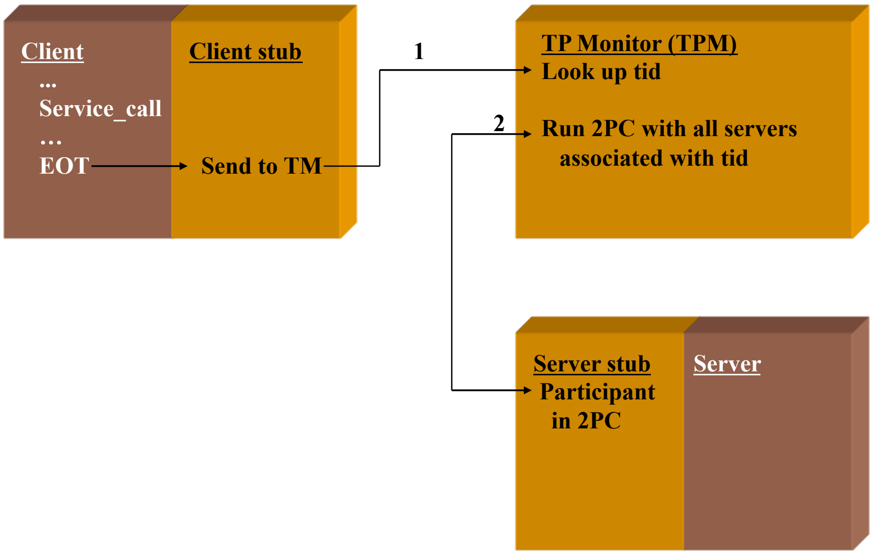


Client
…
Service_call
…
EOT

Client stub
Send to TM

TP Monitor (TPM))
Look up tid

Basic TRPC (committing calls)



Client
…
Service_call
…
EOT

Client stub
Send to TM

1

TP Monitor (TPM)
Look up tid

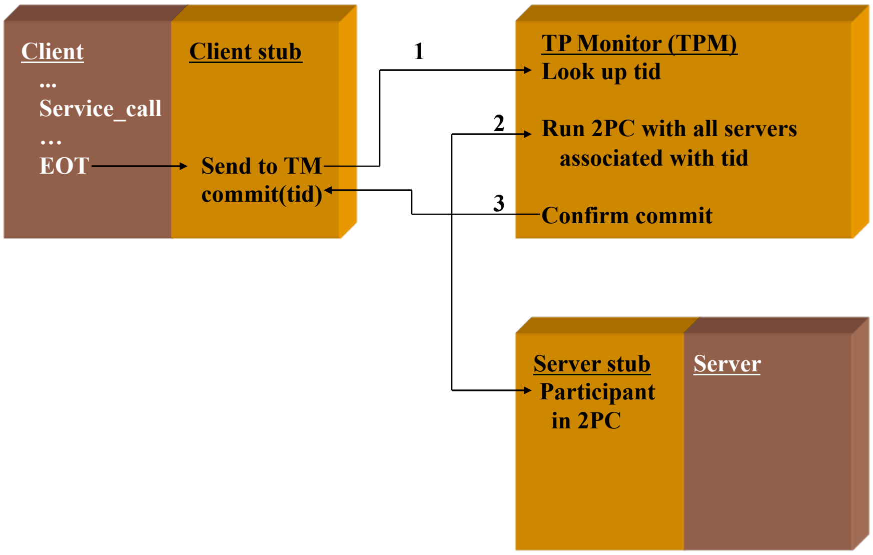
2

Run 2PC with all servers associated with tid

Server stub
Participant in 2PC

Server

Basic TRPC (committing calls)



Client
…
Service_call
…
EOT

Client stub
Send to TM
commit(tid)

1

TP Monitor (TPM)
Look up tid

2

Run 2PC with all servers associated with tid

3

Confirm commit

Server stub
Participant in 2PC

Server

TWO PHASE COMMIT (2PC)

Atomic Commitment

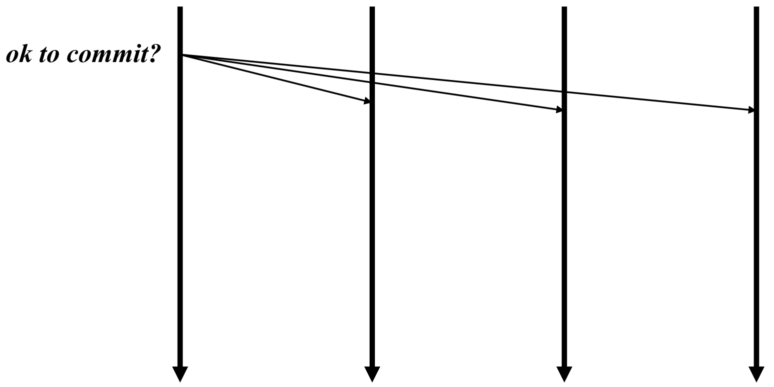
- Given a set of processes
- **Coordinator** (i.e. TPM) wants to initiate an action (commit)
- **Participants** may vote for or against the action
- Perform the action only if all vote in favor
- Otherwise abort
- Goal is *all-or-nothing* outcome

Non-triviality

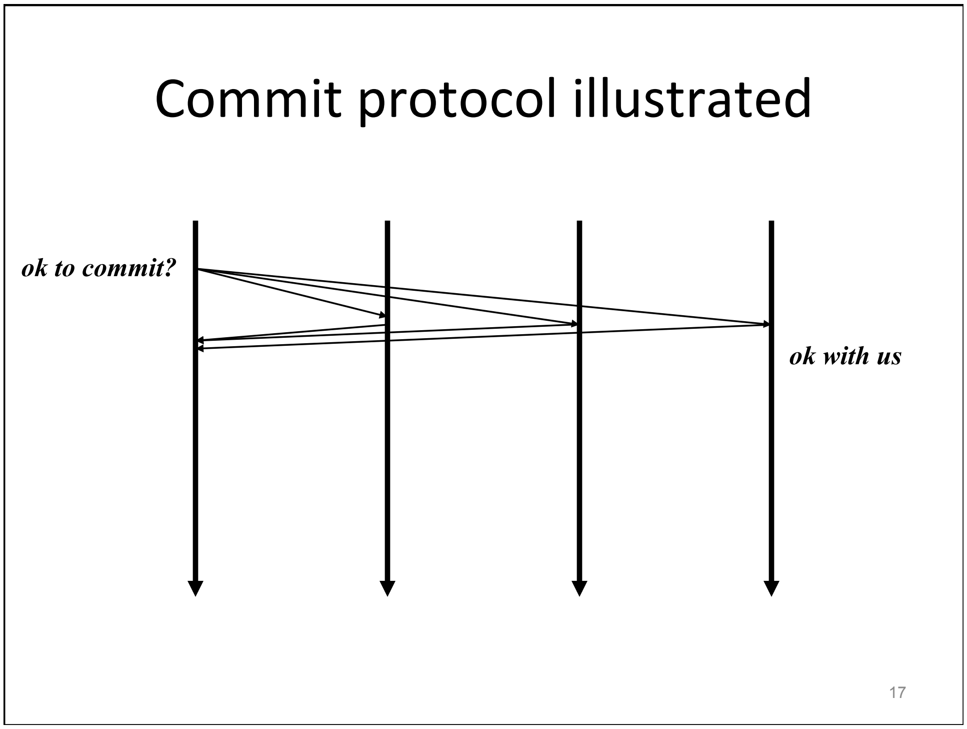
- Avoid solutions that do nothing
- What is a trivial solution?
- What is a validity condition?

15

Commit protocol illustrated

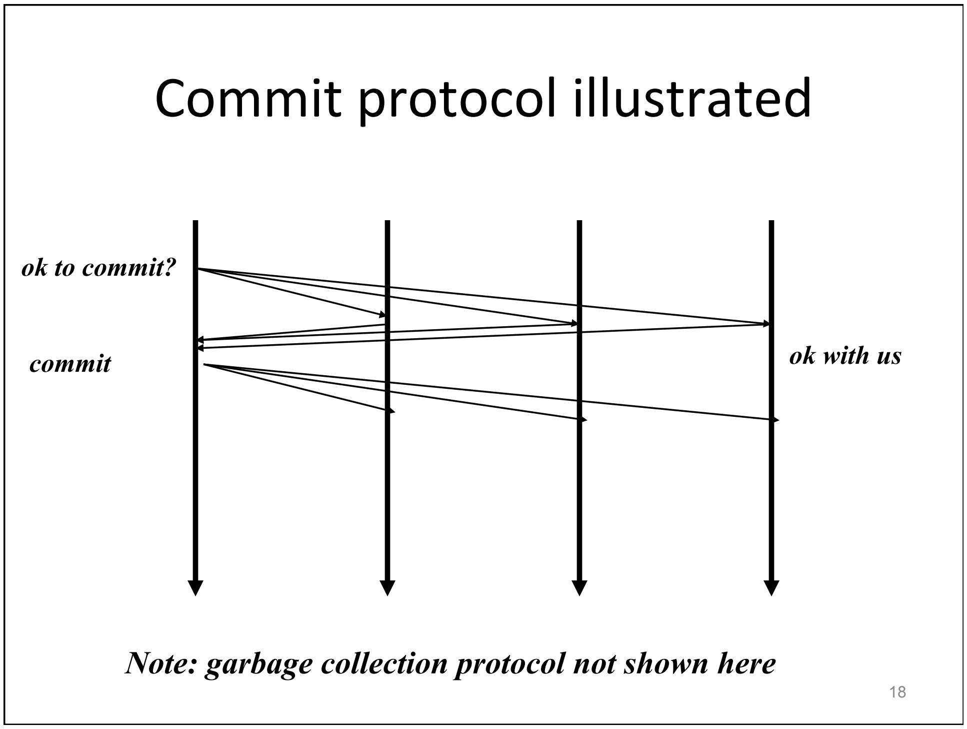


16



Commit protocol illustrated

17



Commit protocol illustrated

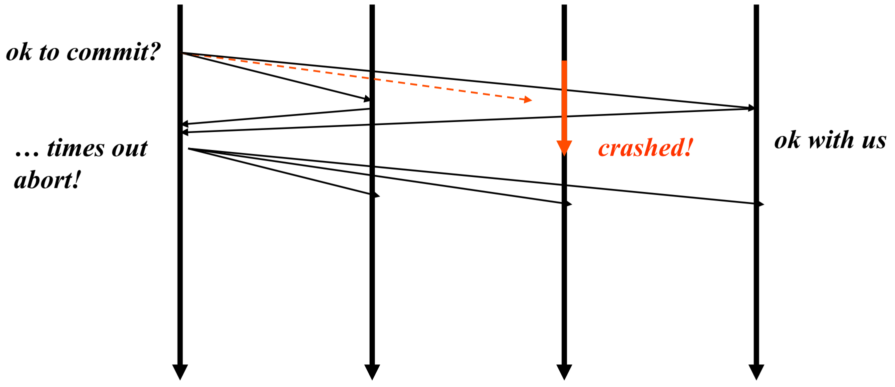
Note: garbage collection protocol not shown here

18

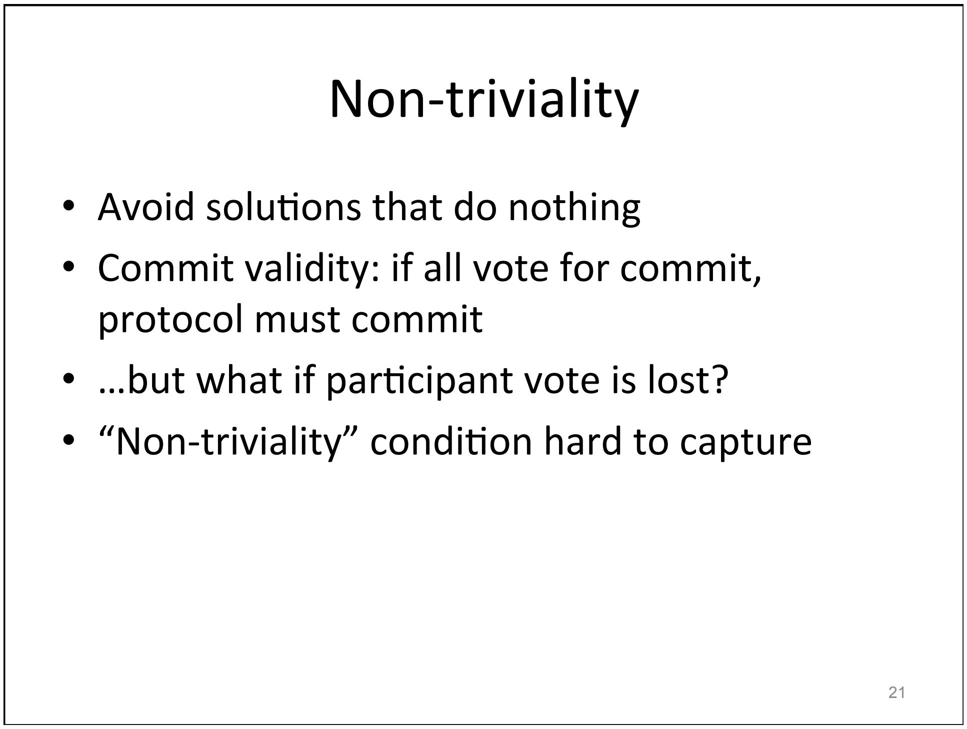
Two-phase Commit

- Phase 0: Flush caches on Web, app server
- Phase 1: Coordinator asks participants for vote
  - Data managers force updates to the log
  - Then say "ok to commit"
- Phase 2: If all are ok to commit, then coordinator tells participants to commit. Otherwise, abort.
- Data managers then make updates permanent or rollback to old values, and release locks

Commit protocol illustrated

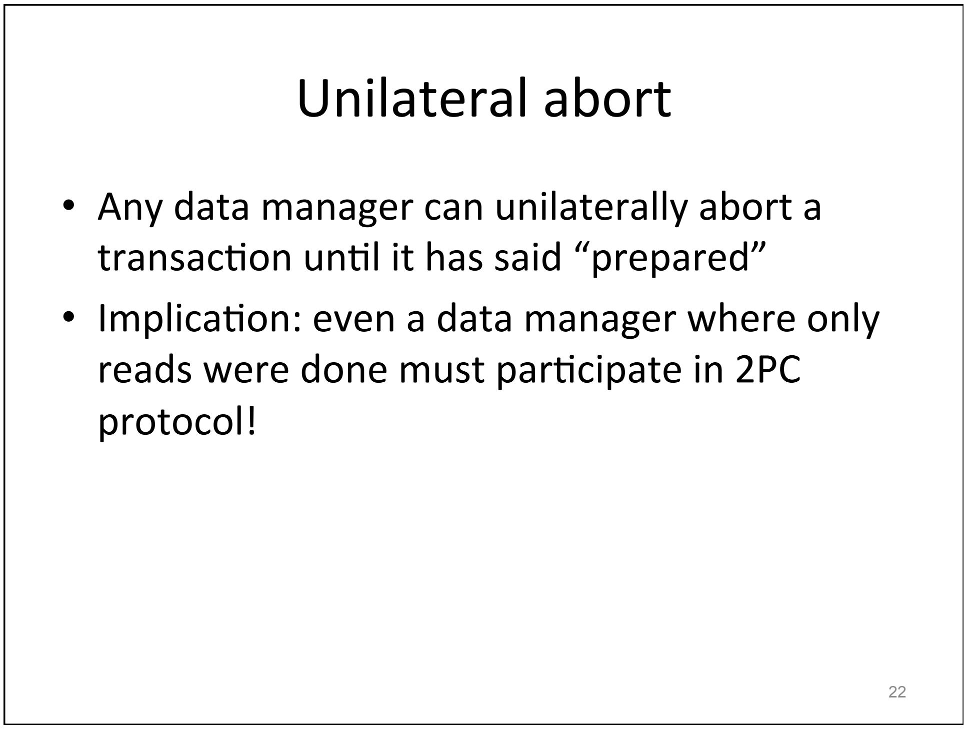


Note: garbage collection protocol not shown here



Non-triviality

- Avoid solutions that do nothing
- Commit validity: if all vote for commit, protocol must commit
- …but what if participant vote is lost?
- “Non-triviality” condition hard to capture



Unilateral abort

- Any data manager can unilaterally abort a transaction until it has said “prepared”
- Implication: even a data manager where only reads were done must participate in 2PC protocol!

PROBLEMS WITH 2PC

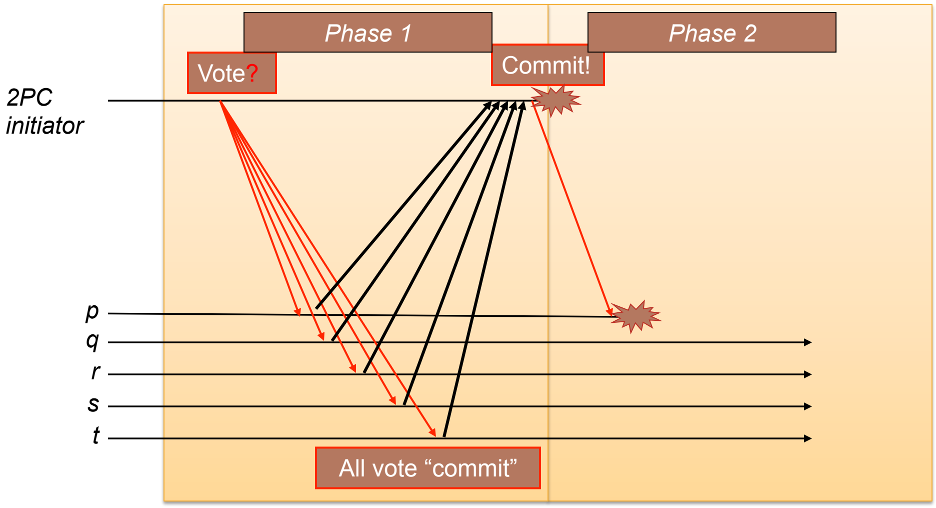
Non-blocking Commit

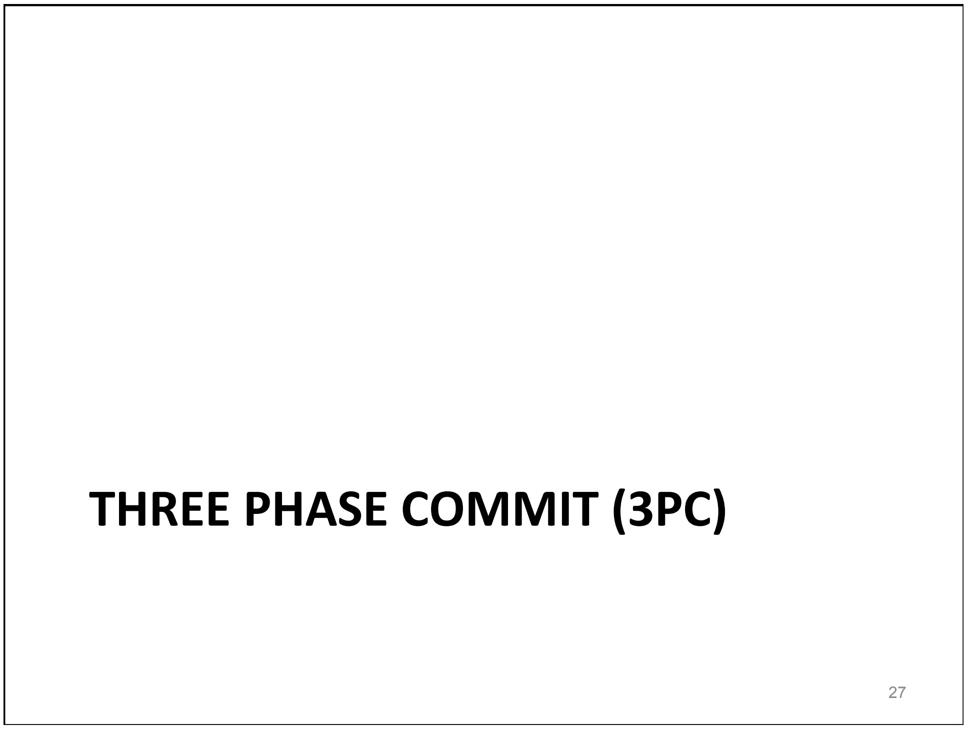
- Goal: a protocol that allows all operational processes to terminate the protocol even if some subset crash

Commit with unreliable failure detectors

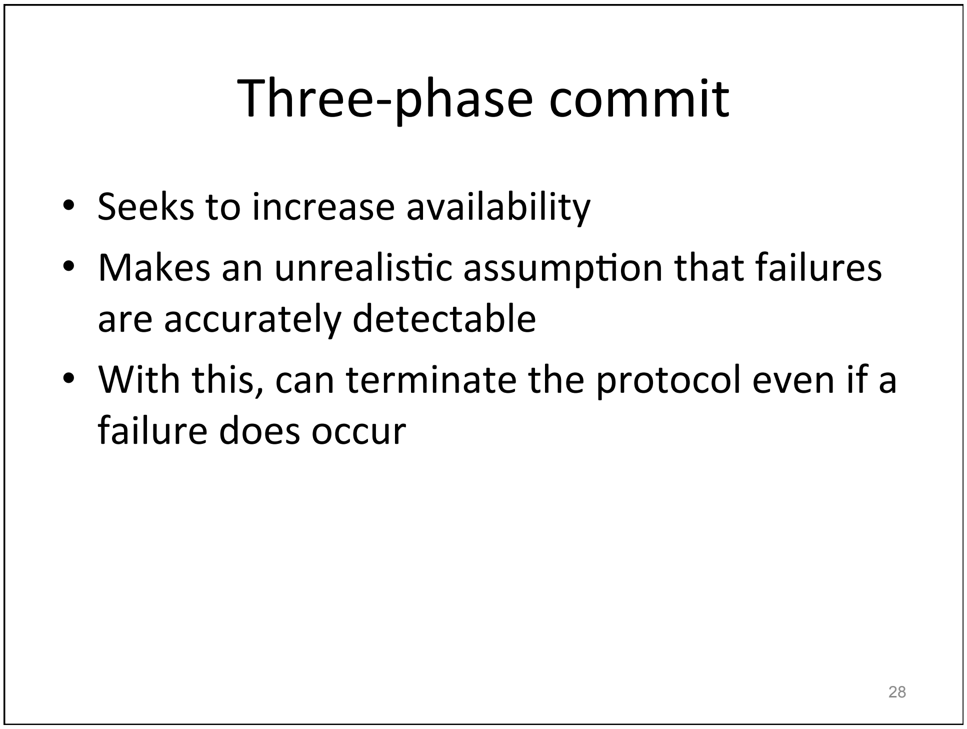
- Assume processes fail by crashing
  - No Byzantine failures
- Coordinator detects failures (unreliably) using timouts
- Challenge: terminate the protocol if the coordinator fails

2 phase commit





THREE PHASE COMMIT (3PC)



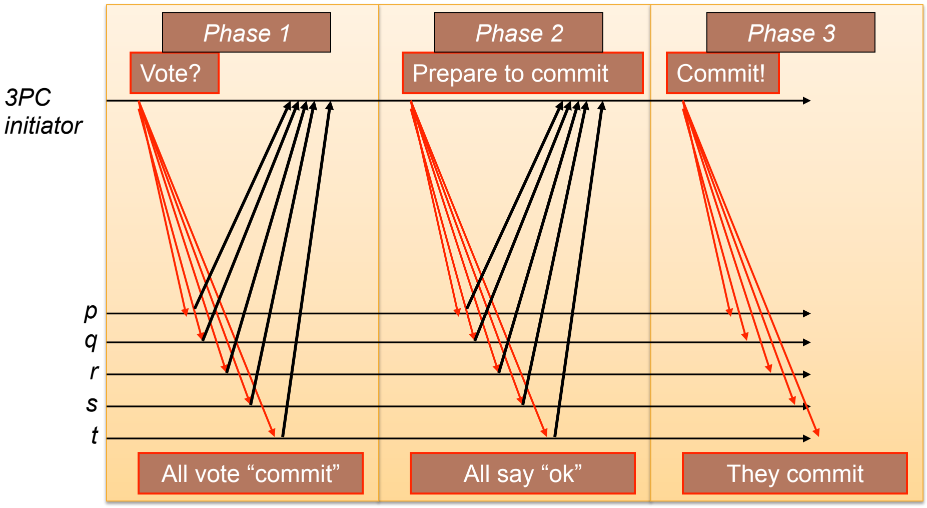
Three-phase commit

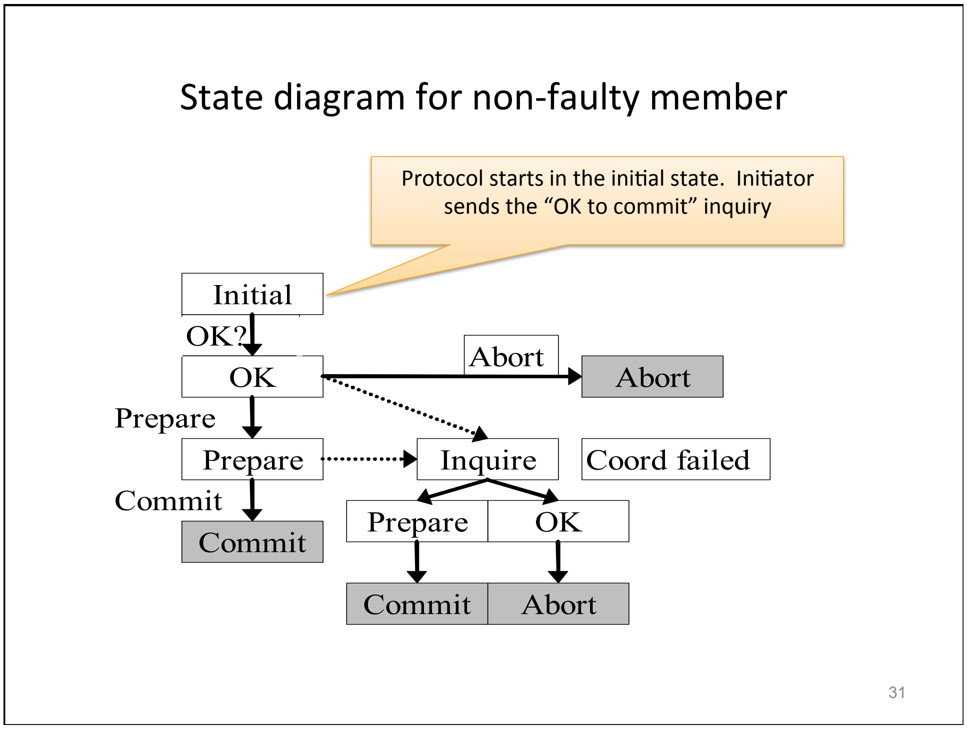
- Seeks to increase availability
- Makes an unrealistic assumption that failures are accurately detectable
- With this, can terminate the protocol even if a failure does occur

Three-phase commit

- Coordinator starts protocol by sending request
- Participants vote to commit or to abort
- Coordinator collects votes, decides on outcome
- Coordinator can abort immediately
- To commit, coordinator first sends a "prepare to commit" message
- Participants acknowledge, commit occurs during a final round of "commit" messages

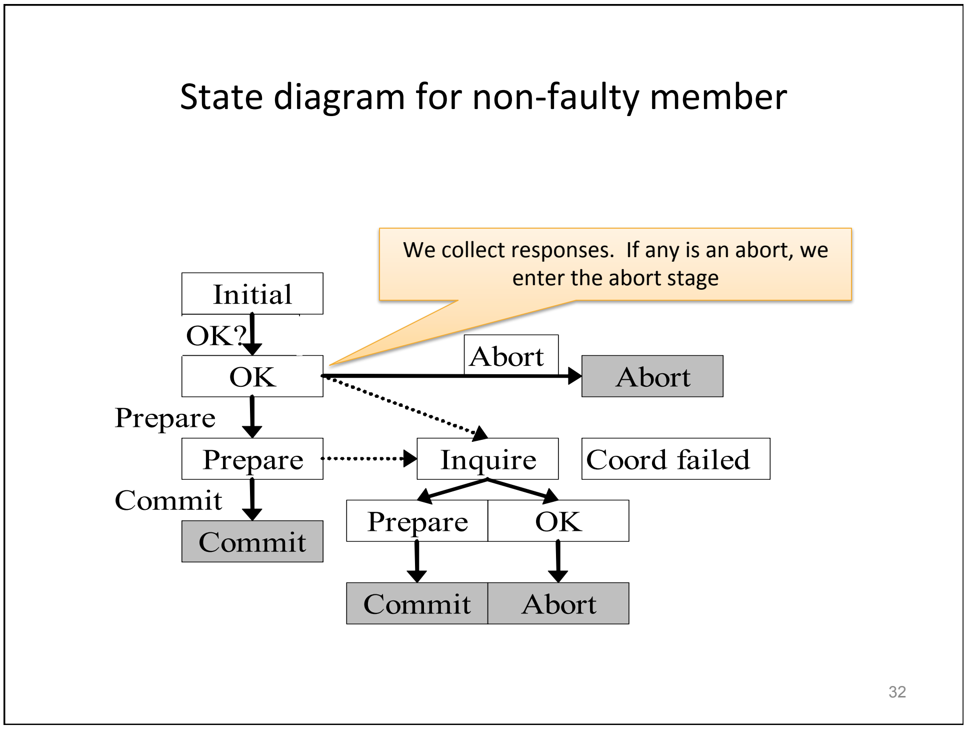
Three-phase commit





State diagram for non-faulty member

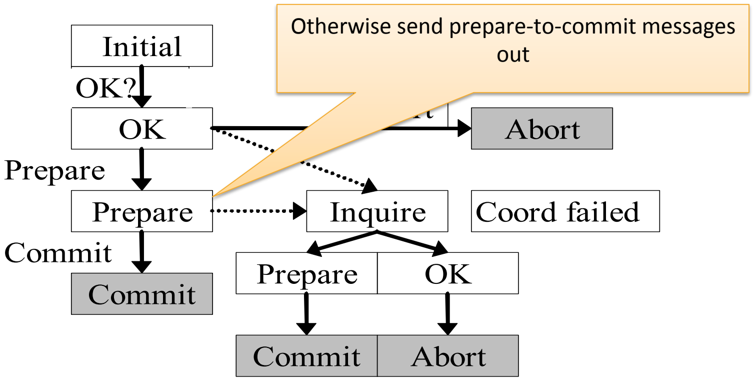
Protocol starts in the initial state. Initiator sends the "OK to commit" inquiry



State diagram for non-faulty member

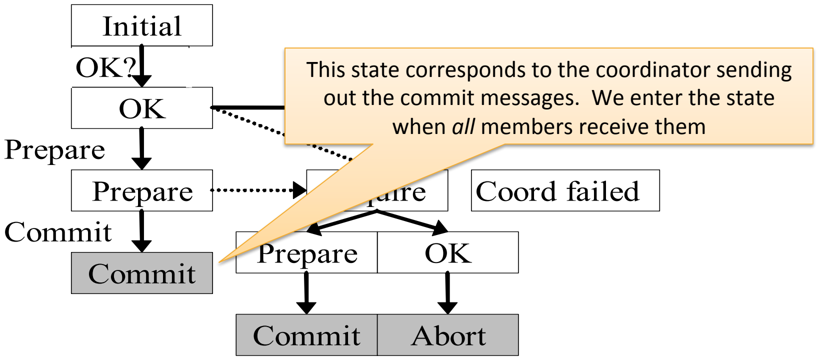
We collect responses. If any is an abort, we enter the abort stage

State diagram for non-faulty member



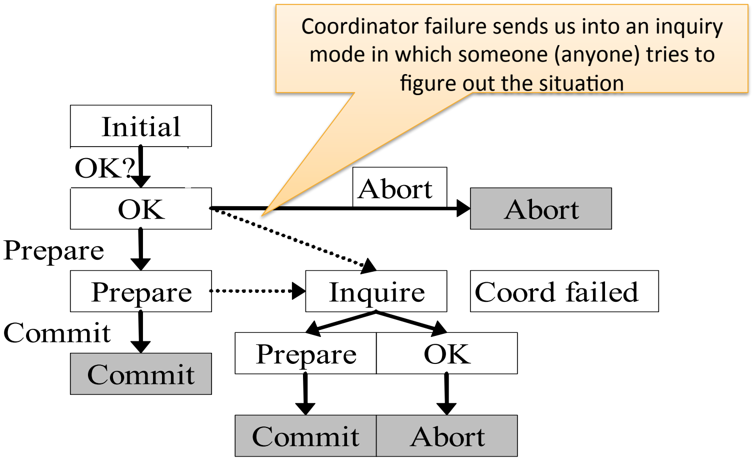
Otherwise send prepare-to-commit messages out

State diagram for non-faulty member



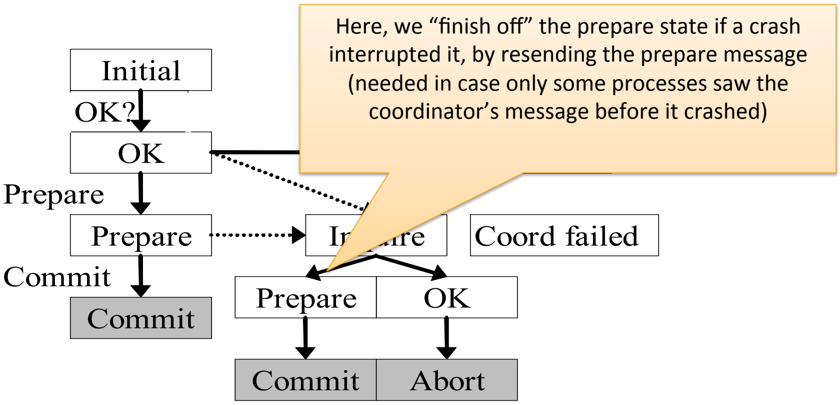
This state corresponds to the coordinator sending out the commit messages. We enter the state when *all* members receive them

State diagram for non-faulty member



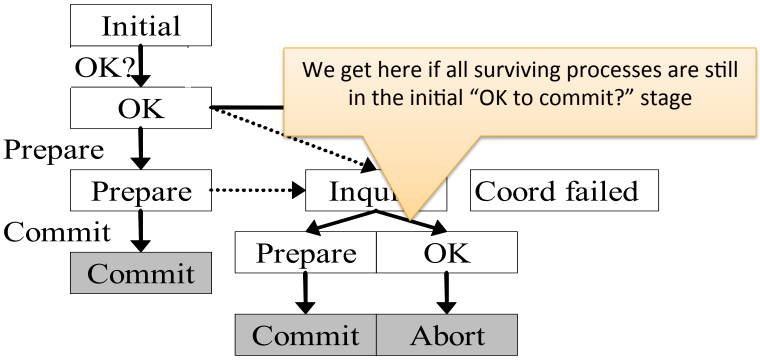
Coordinator failure sends us into an inquiry mode in which someone (anyone) tries to figure out the situation

State diagram for non-faulty member



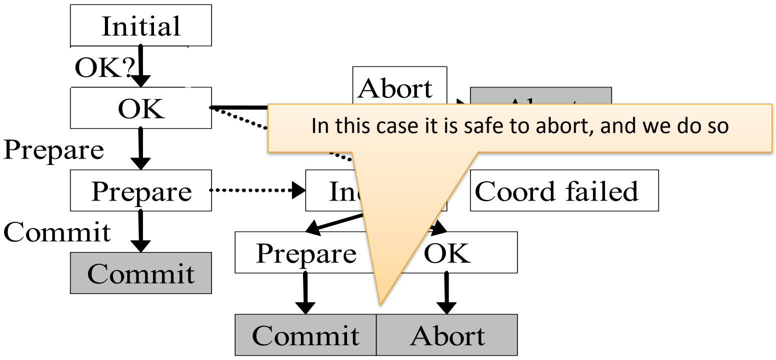
Here, we "finish off" the prepare state if a crash interrupted it, by resending the prepare message (needed in case only some processes saw the coordinator's message before it crashed)

State diagram for non-faulty member



We get here if all surviving processes are still in the initial "OK to commit?" stage

State diagram for non-faulty member



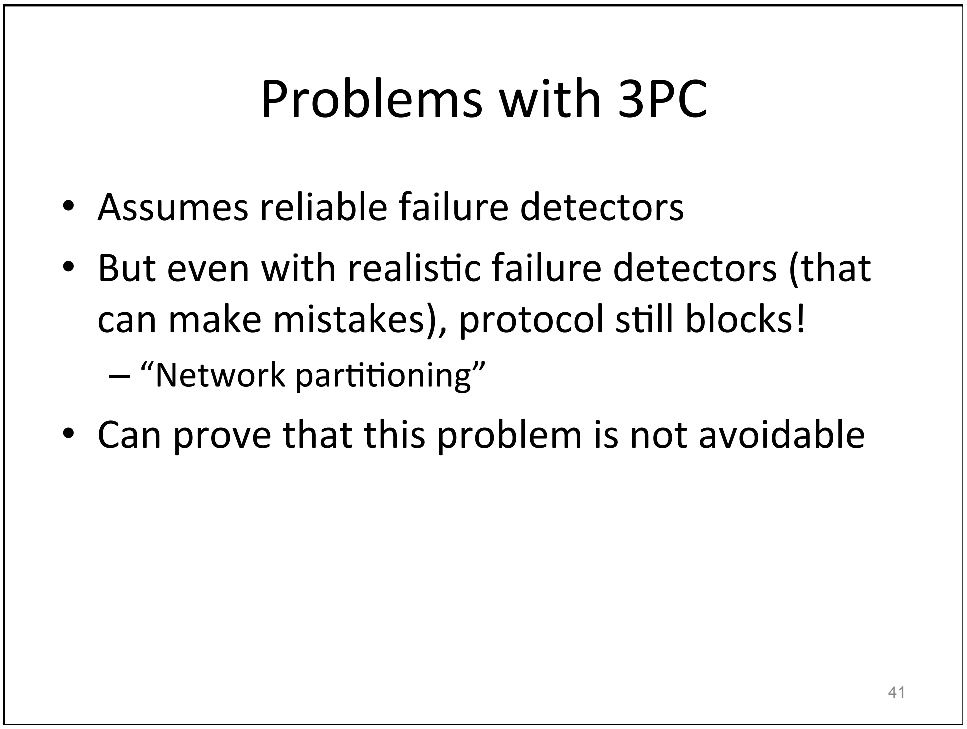
In this case it is safe to abort, and we do so

Observations about 3PC

- Key point: Extra buffer state
- What if none of surviving participants have heard from coordinator?
  - After voting phase
  - 2PC: Some crashed processes may have committed
  - 3PC: No crashed process has committed yet (Why?)

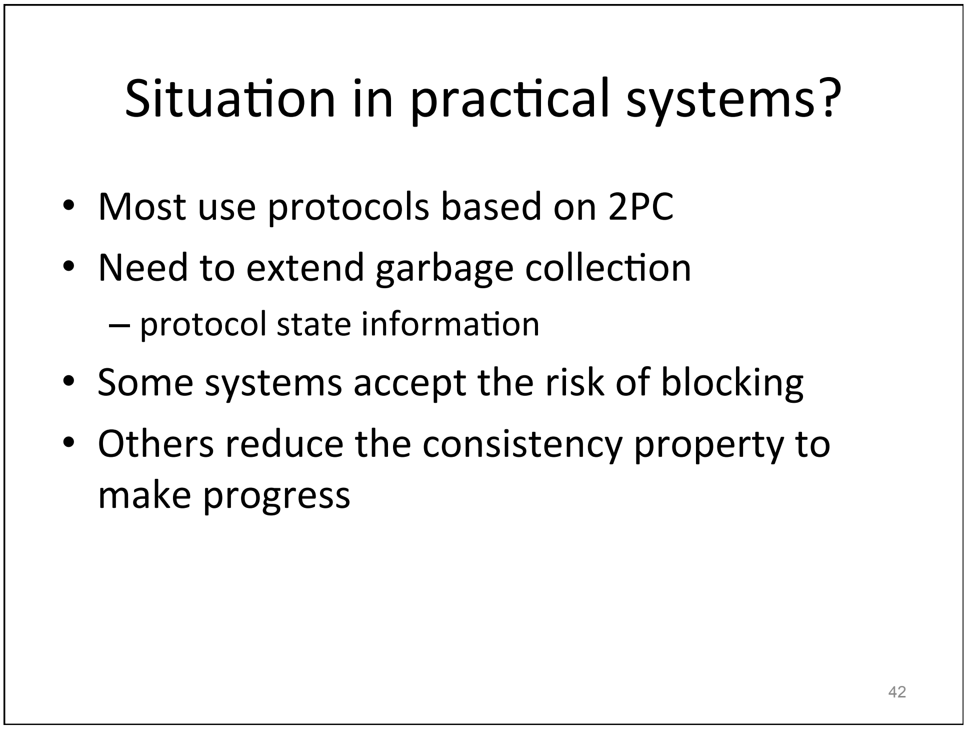
Observations about 3PC

- If any process is in “prepare to commit” all voted for commit
- Protocol commits only when all surviving processes have acknowledged prepare to commit
- After coordinator fails, it is easy to run the protocol forward to commit state (or back to abort state)



Problems with 3PC

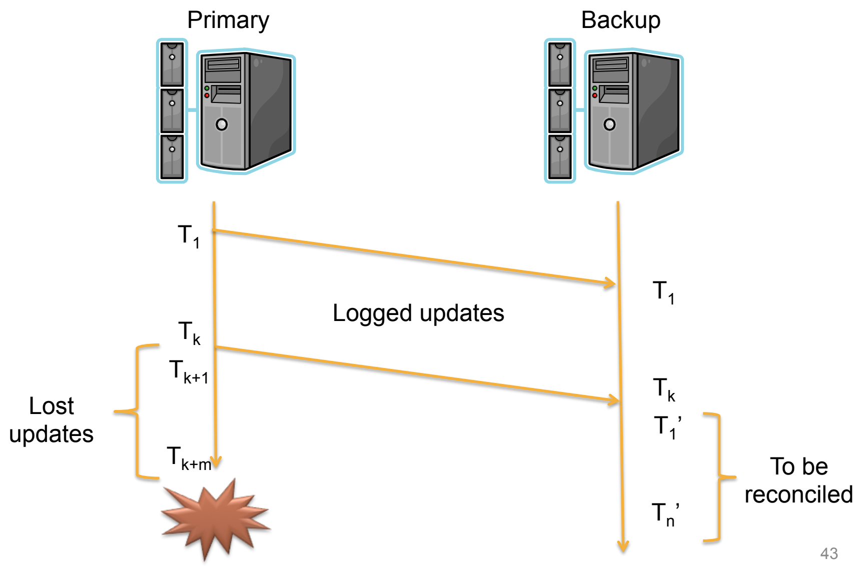
- Assumes reliable failure detectors
- But even with realistic failure detectors (that can make mistakes), protocol still blocks!
  - "Network partitioning"
- Can prove that this problem is not avoidable



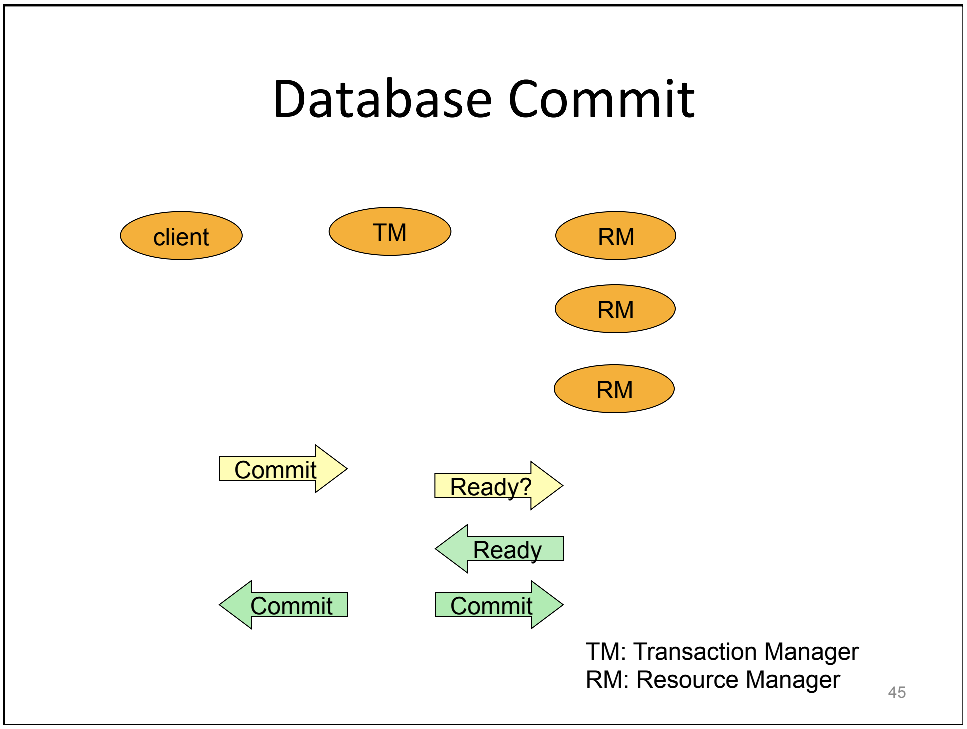
Situation in practical systems?

- Most use protocols based on 2PC
- Need to extend garbage collection
  - protocol state information
- Some systems accept the risk of blocking
- Others reduce the consistency property to make progress

Example: Primary with Backup



PAXOS AND ATOMIC COMMIT



Database Commit

client

TM

RM

RM

RM

Commit

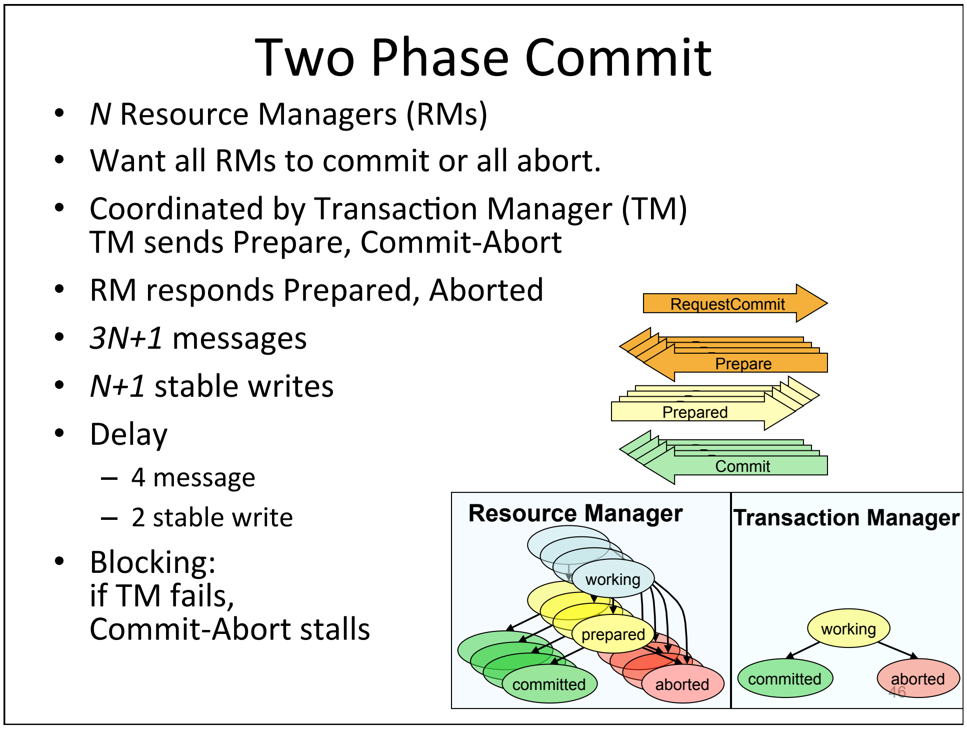
Ready?

Ready

Commit

Commit

TM: Transaction Manager
RM: Resource Manager



Two Phase Commit

- *N* Resource Managers (RMs)
- Want all RMs to commit or all abort.
- Coordinated by Transaction Manager (TM)
  TM sends Prepare, Commit-Abort
- RM responds Prepared, Aborted
- *3N+1* messages
- *N+1* stable writes
- Delay
  - 4 message
  - 2 stable write
- Blocking:
  if TM fails,
  Commit-Abort stalls

RequestCommit

Prepare

Prepared

Commit

Resource Manager

working

prepared

committed

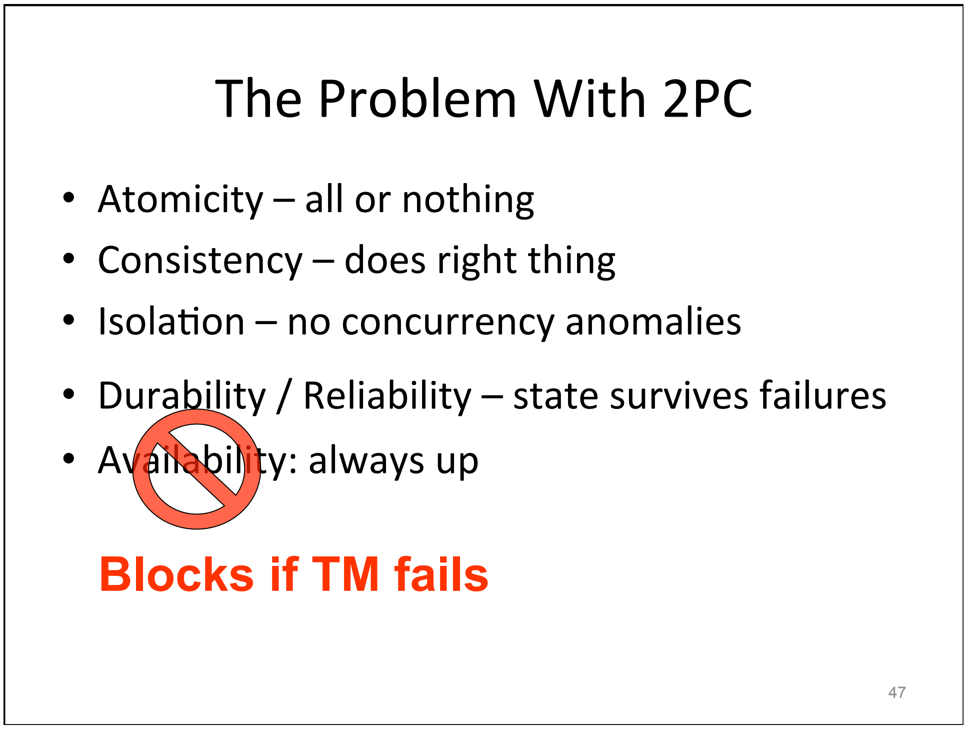
aborted

Transaction Manager

working

committed

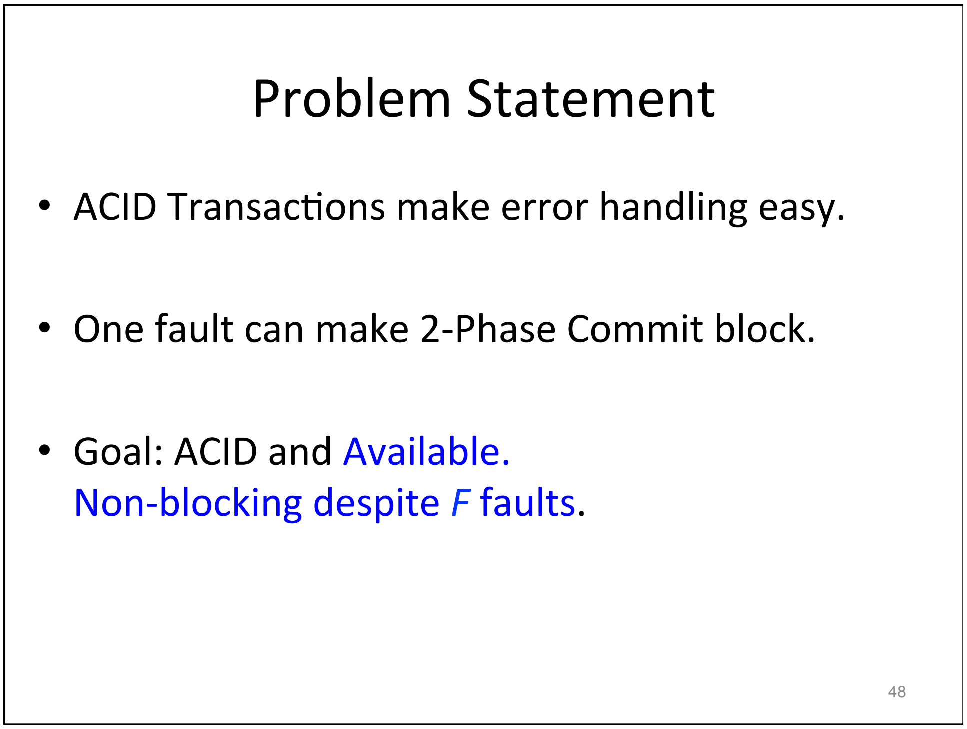
aborted



The Problem With 2PC

- Atomicity – all or nothing
- Consistency – does right thing
- Isolation – no concurrency anomalies
- Durability / Reliability – state survives failures
- 🚫 Availability: always up

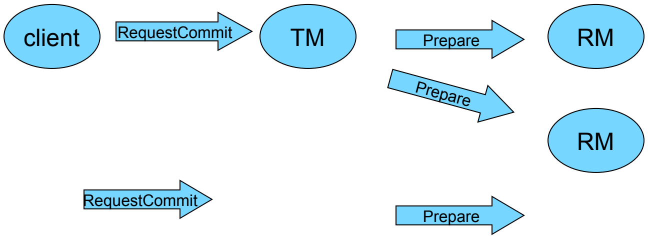
Blocks if TM fails



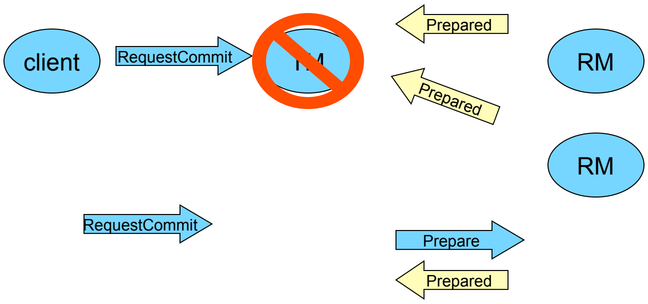
Problem Statement

- ACID Transactions make error handling easy.
- One fault can make 2-Phase Commit block.
- Goal: ACID and Available.
  Non-blocking despite *F* faults.

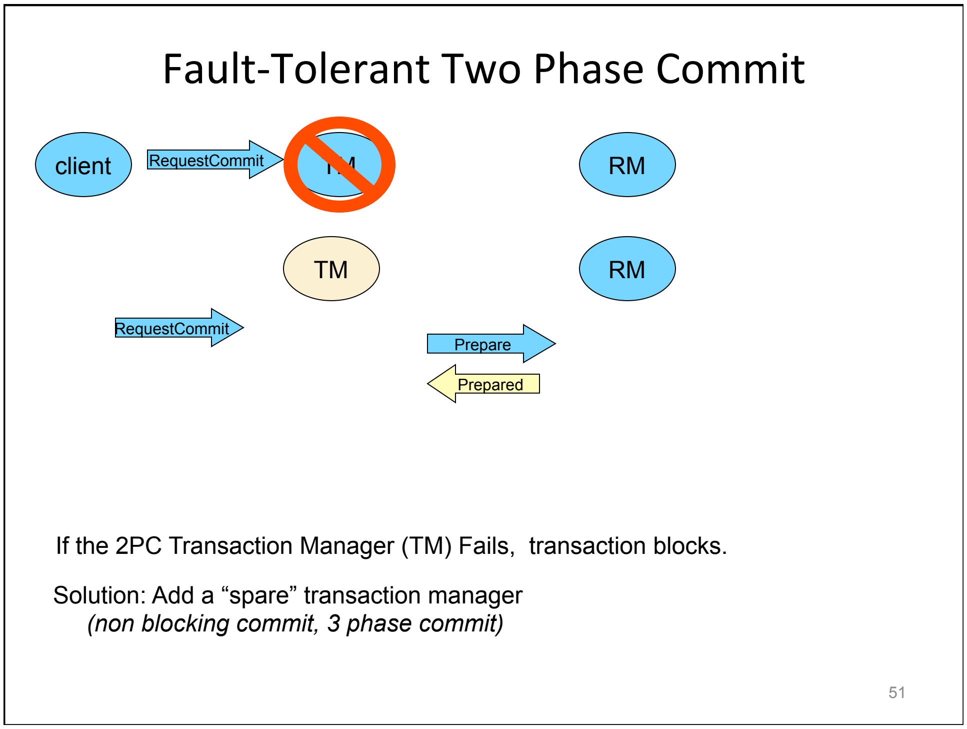
Fault-Tolerant Two Phase Commit



Fault-Tolerant Two Phase Commit



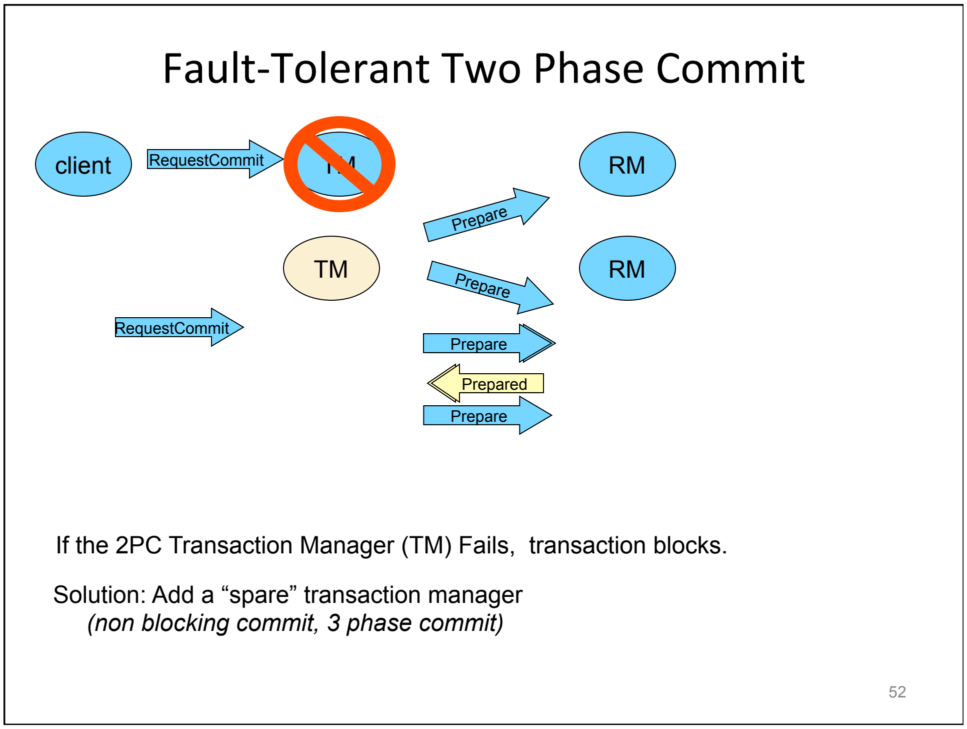
If the 2PC Transaction Manager (TM) Fails, transaction blocks.



Fault-Tolerant Two Phase Commit

If the 2PC Transaction Manager (TM) Fails, transaction blocks.

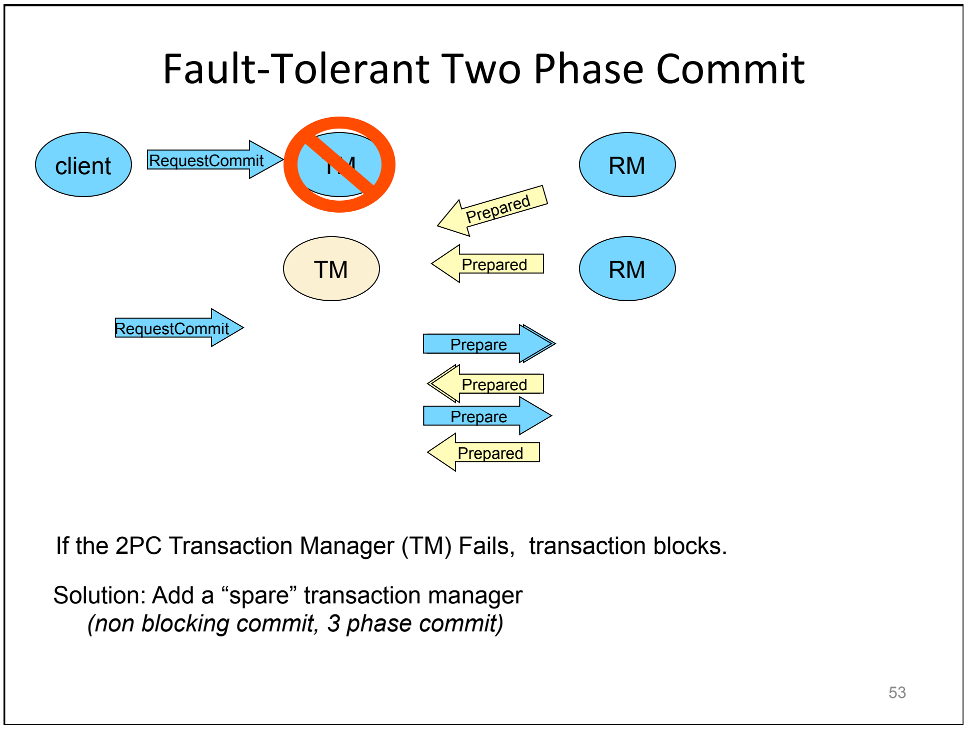
Solution: Add a “spare” transaction manager
(non blocking commit, 3 phase commit)



Fault-Tolerant Two Phase Commit

If the 2PC Transaction Manager (TM) Fails, transaction blocks.

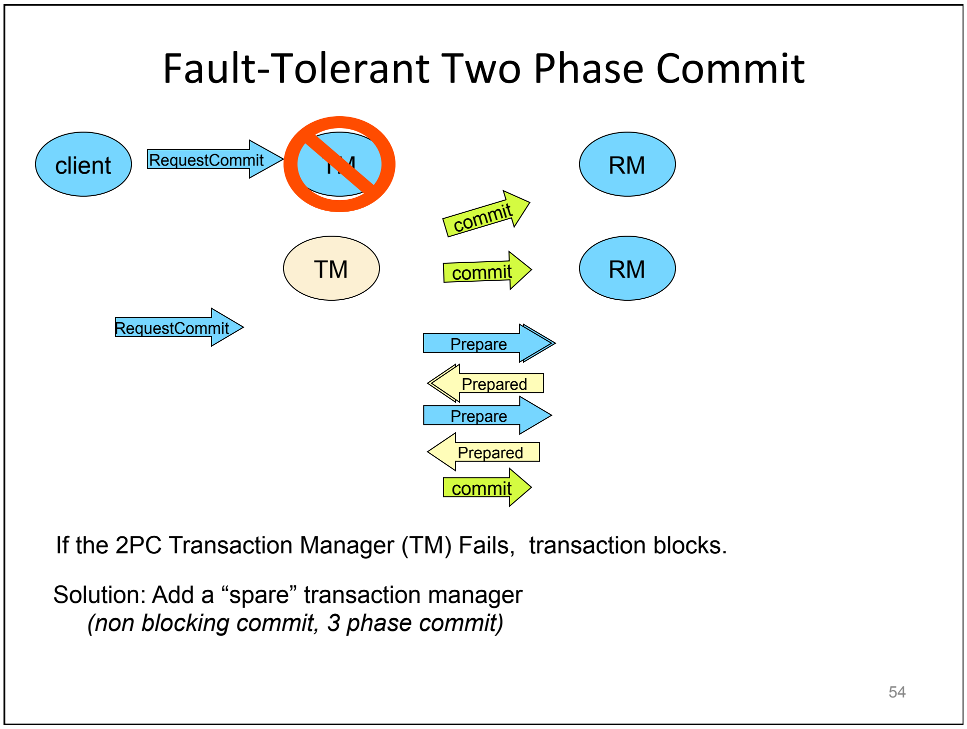
Solution: Add a “spare” transaction manager
(non blocking commit, 3 phase commit)



Fault-Tolerant Two Phase Commit

If the 2PC Transaction Manager (TM) Fails, transaction blocks.

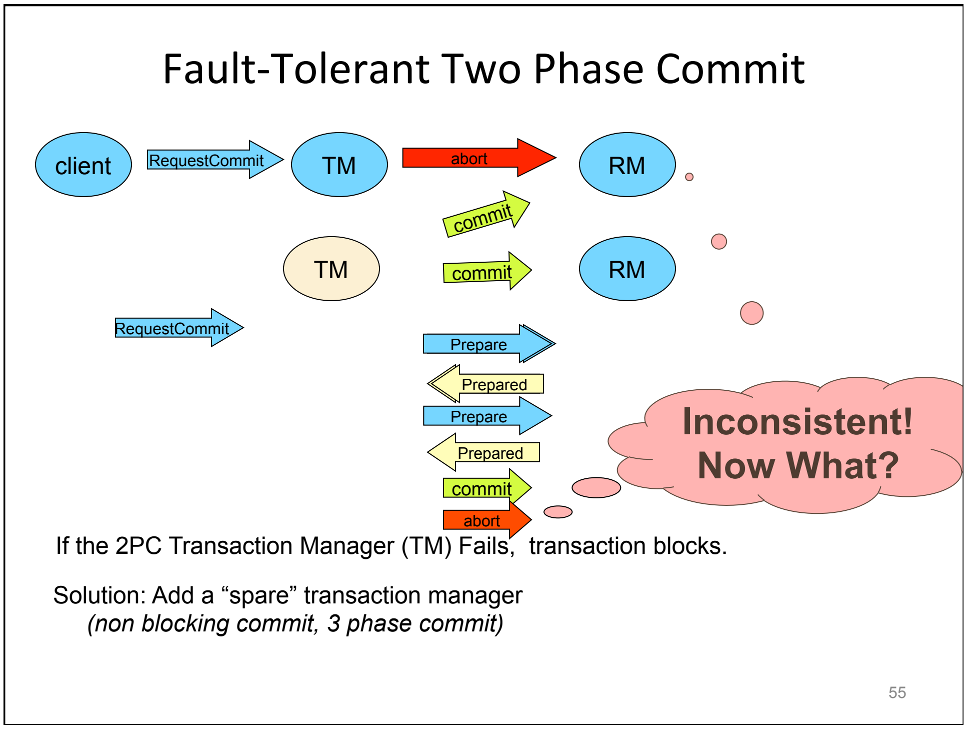
Solution: Add a “spare” transaction manager
(non blocking commit, 3 phase commit)



Fault-Tolerant Two Phase Commit

If the 2PC Transaction Manager (TM) Fails, transaction blocks.

Solution: Add a “spare” transaction manager
(non blocking commit, 3 phase commit)



Fault-Tolerant Two Phase Commit

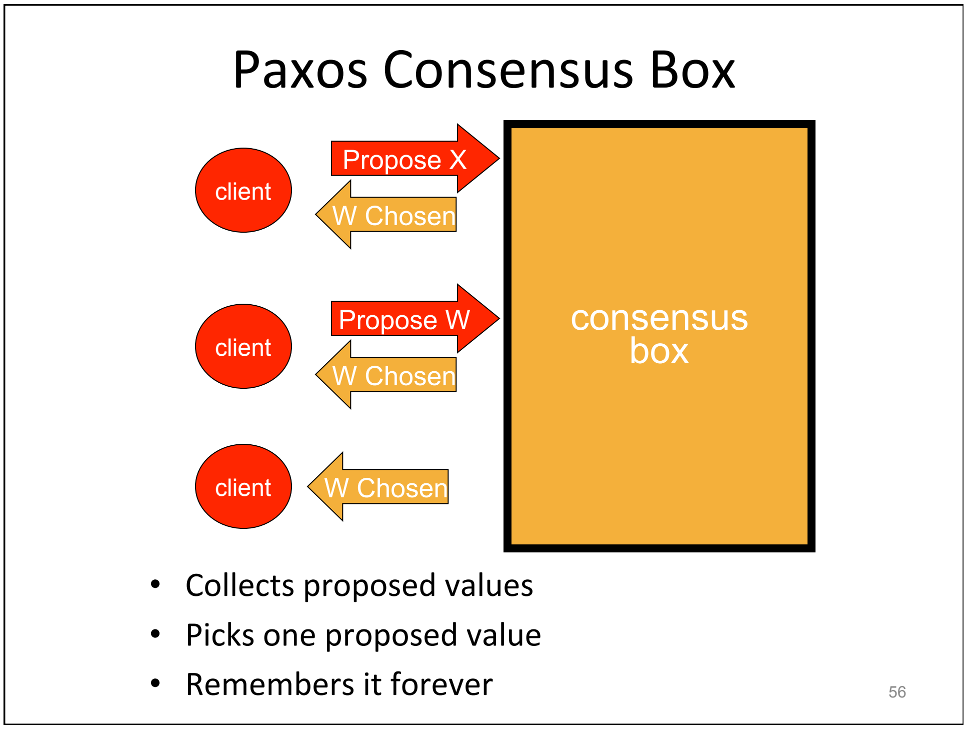
client
RequestCommit
TM
abort
RM
commit
TM
commit
RM
RequestCommit
Prepare
Prepared
Prepare
Prepared
commit
abort

Inconsistent!
Now What?

If the 2PC Transaction Manager (TM) Fails, transaction blocks.

Solution: Add a “spare” transaction manager
(non blocking commit, 3 phase commit)

55



Paxos Consensus Box

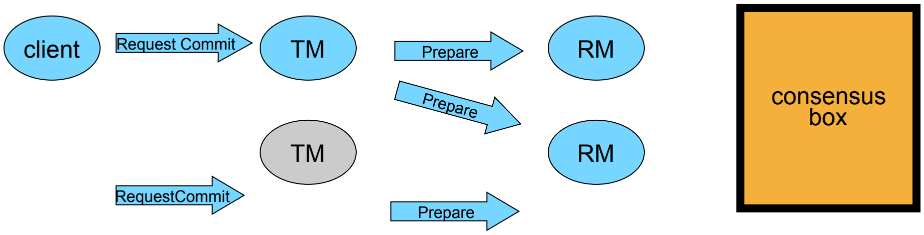
client
Propose X
W Chosen
client
Propose W
W Chosen
client
W Chosen

consensus
box

- Collects proposed values
- Picks one proposed value
- Remembers it forever

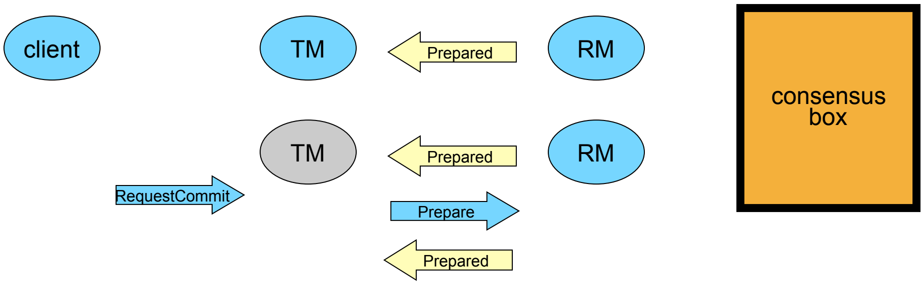
56

Consensus for Commit
The Obvious Approach

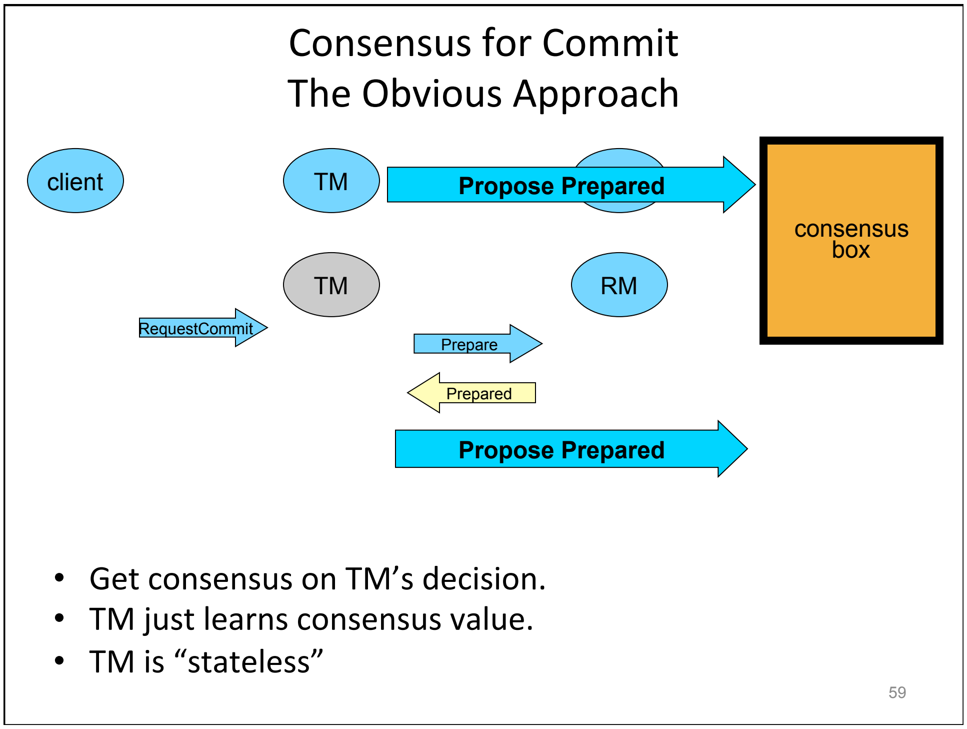


- Get consensus on TM's decision.
- TM just learns consensus value.
- TM is "stateless"

Consensus for Commit
The Obvious Approach



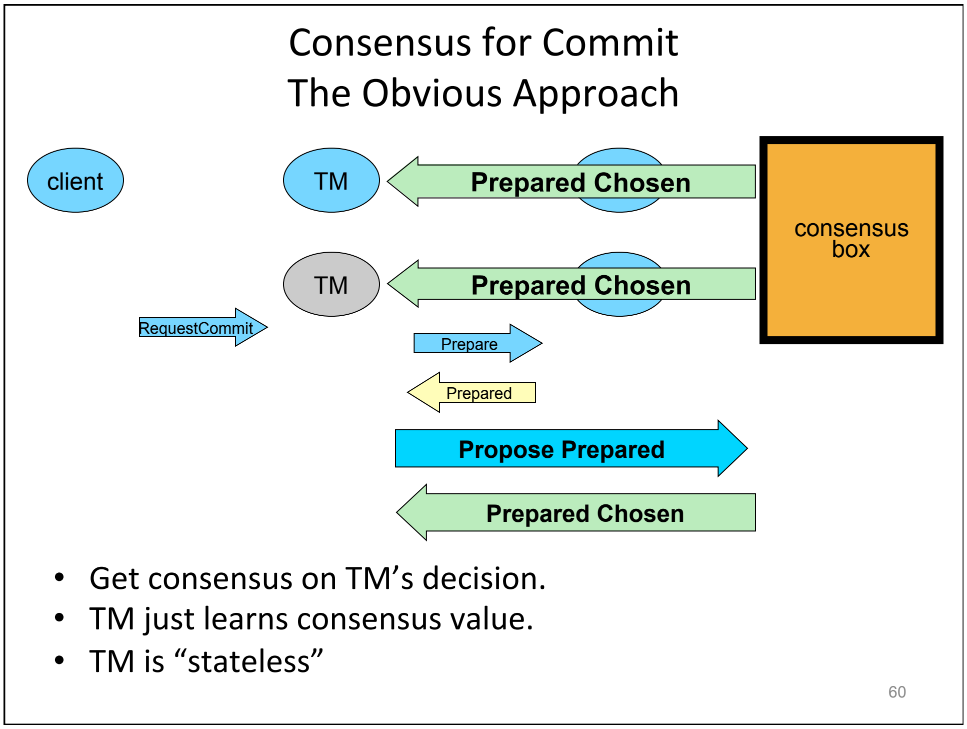
- Get consensus on TM's decision.
- TM just learns consensus value.
- TM is "stateless"



Consensus for Commit
The Obvious Approach

- Get consensus on TM’s decision.
- TM just learns consensus value.
- TM is “stateless”

59

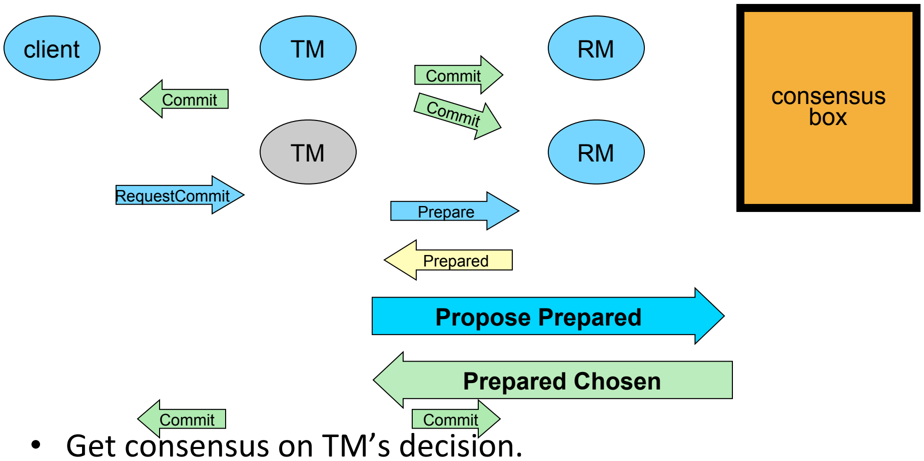


Consensus for Commit
The Obvious Approach

- Get consensus on TM’s decision.
- TM just learns consensus value.
- TM is “stateless”

60

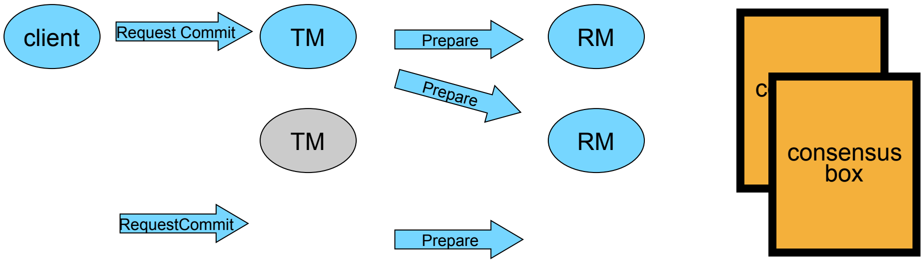
Consensus for Commit
The Obvious Approach



- Get consensus on TM’s decision.
- TM just learns consensus value.
- TM is “stateless”

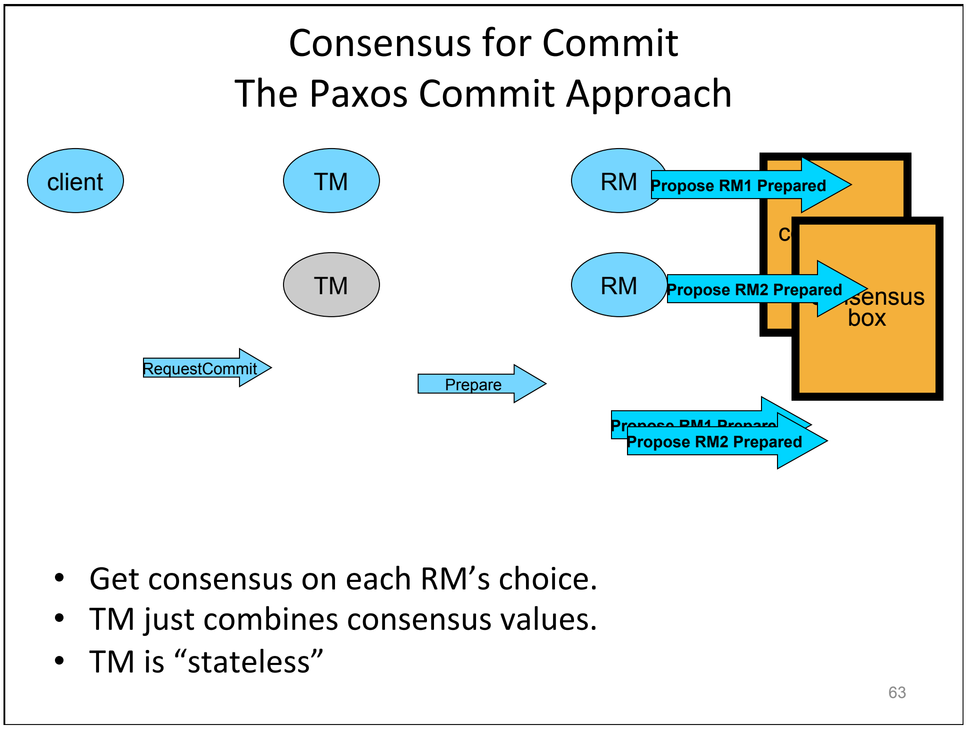
61

Consensus for Commit
The Paxos Commit Approach



- Get consensus on each RM’s choice.
- TM just combines consensus values.
- TM is “stateless”

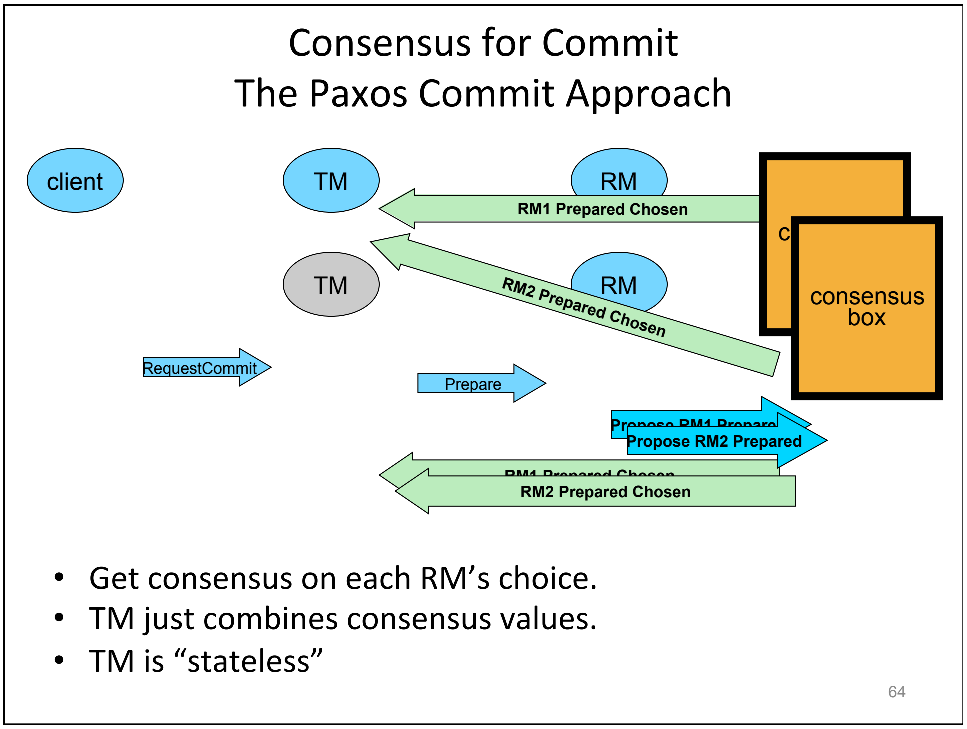
62



Consensus for Commit
The Paxos Commit Approach

- Get consensus on each RM’s choice.
- TM just combines consensus values.
- TM is “stateless”

63

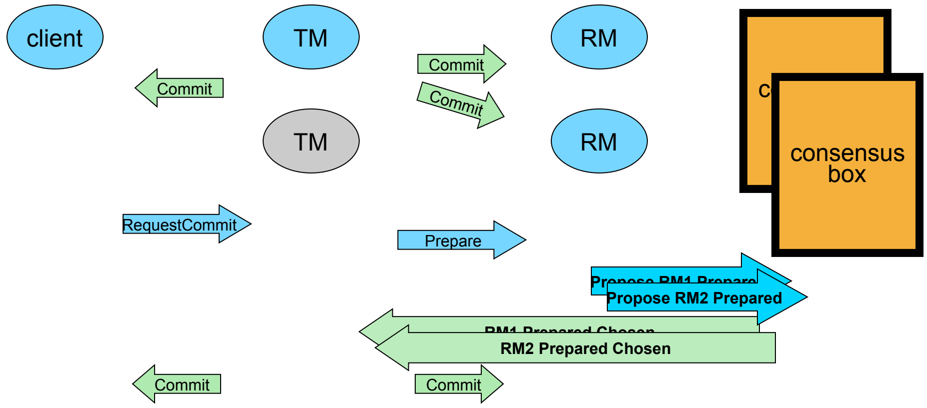


Consensus for Commit
The Paxos Commit Approach

- Get consensus on each RM’s choice.
- TM just combines consensus values.
- TM is “stateless”

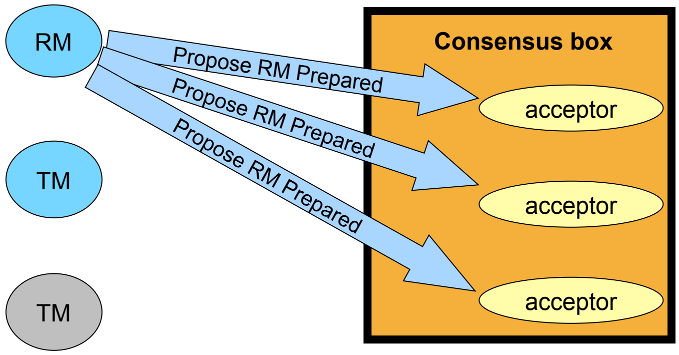
64

Consensus for Commit
The Paxos Commit Approach



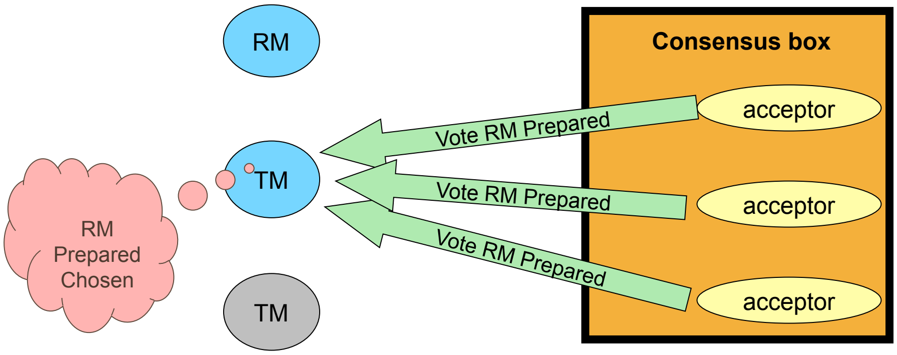
- Get consensus on each RM’s choice.
- TM just combines consensus values.
- TM is “stateless”

Consensus in Action



- The normal (failure-free) case

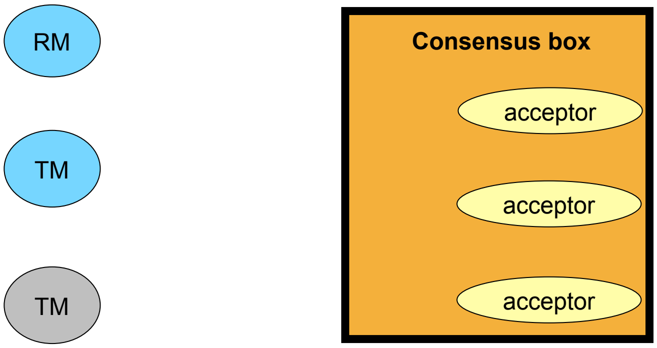
Consensus in Action



- The normal (failure-free) case
- Two message delays
- Can optimize

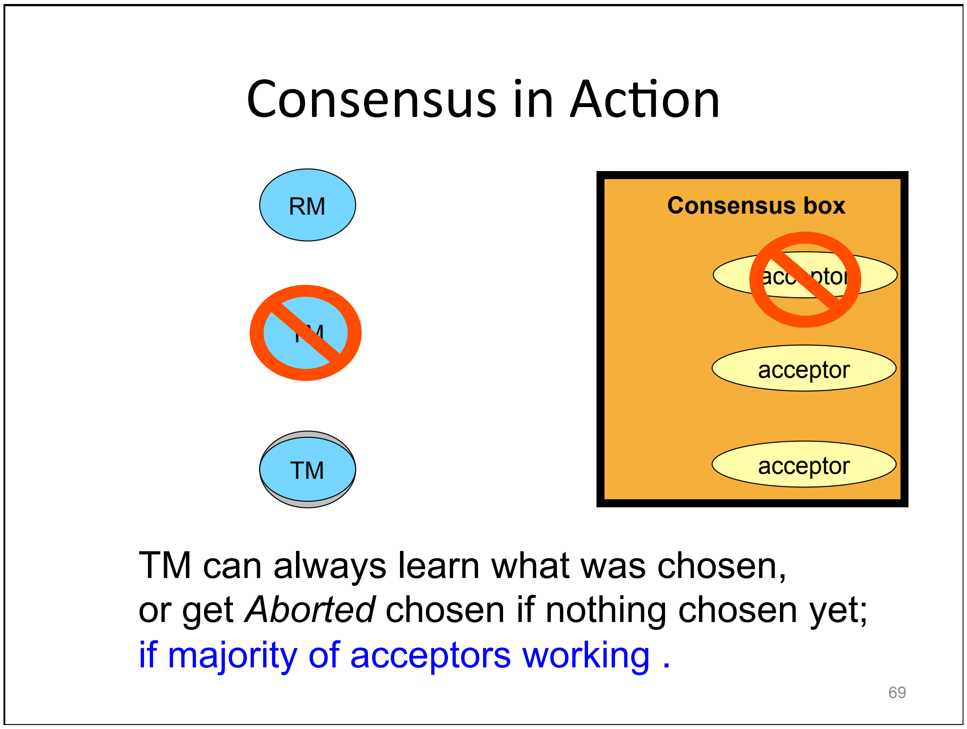
67

Consensus in Action



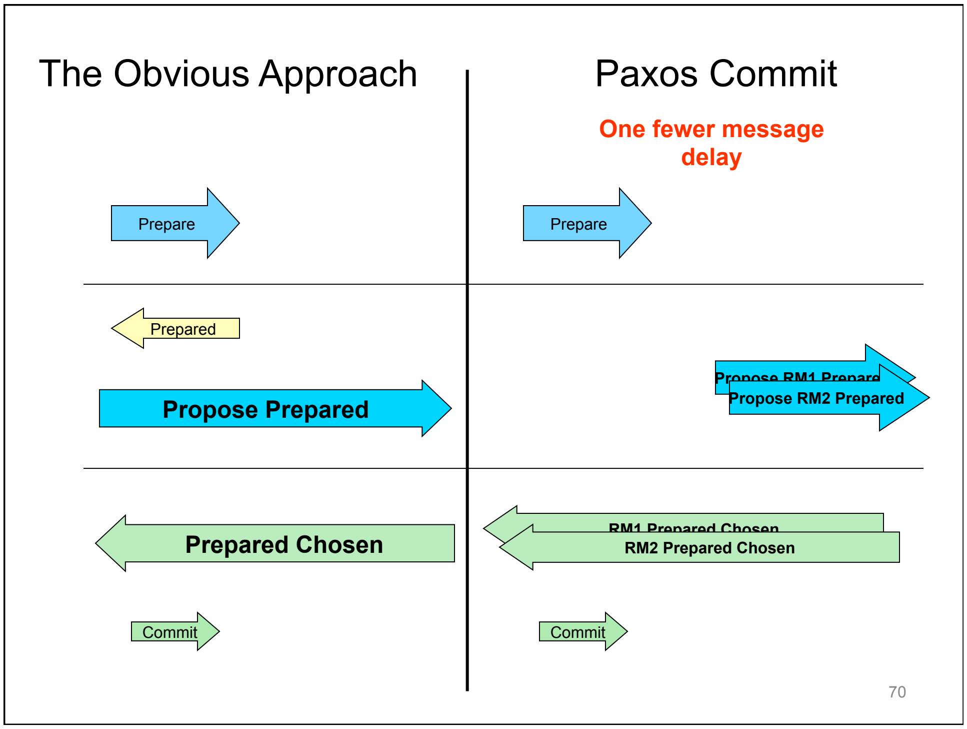
TM can always learn what was chosen,
or get *Aborted* chosen if nothing chosen yet;

68



Consensus in Action

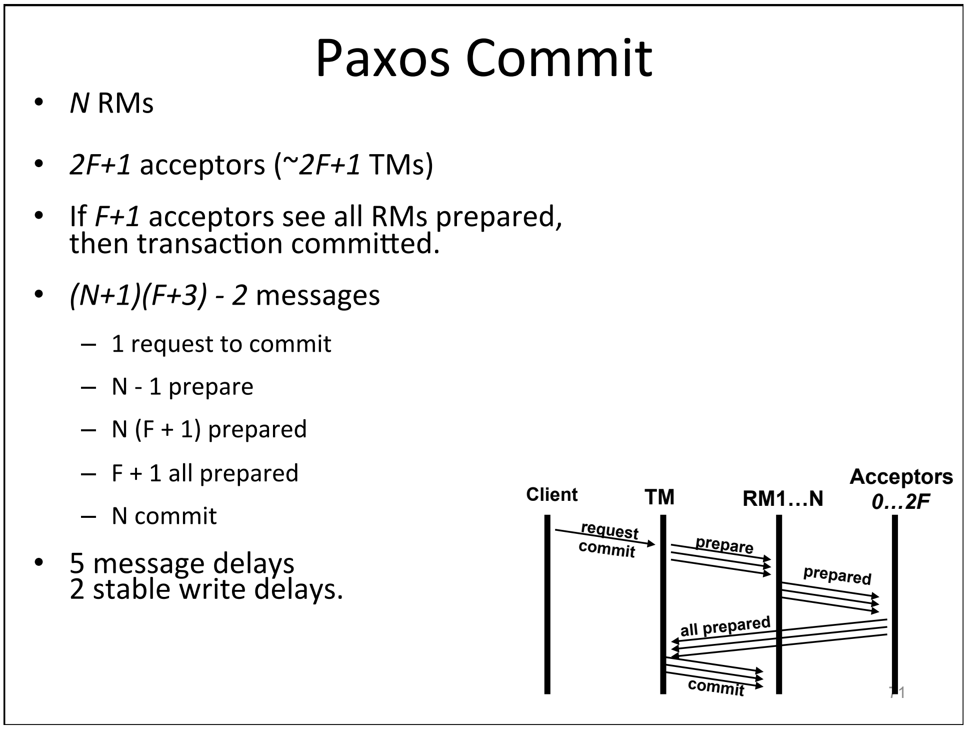
TM can always learn what was chosen,
or get *Aborted* chosen if nothing chosen yet;
if majority of acceptors working .



The Obvious Approach

Paxos Commit

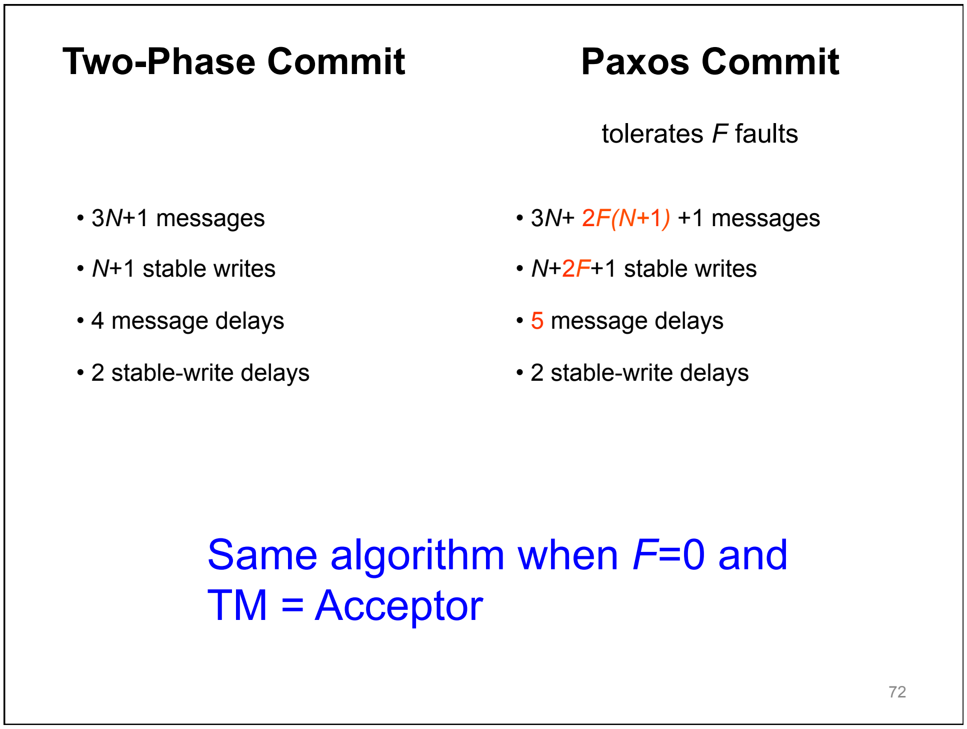
One fewer message delay



Paxos Commit

- N RMs
- $2F+1$ acceptors (~$2F+1$ TMs)
- If $F+1$ acceptors see all RMs prepared, then transaction committed.
- $(N+1)(F+3) - 2$ messages
  - 1 request to commit
  - N - 1 prepare
  - N (F + 1) prepared
  - F + 1 all prepared
  - N commit
- 5 message delays
  2 stable write delays.

71



Two-Phase Commit

- 3N+1 messages
- N+1 stable writes
- 4 message delays
- 2 stable-write delays

Paxos Commit

tolerates F faults

- 3N+ 2F(N+1) +1 messages
- N+2F+1 stable writes
- 5 message delays
- 2 stable-write delays

Same algorithm when F=0 and TM = Acceptor

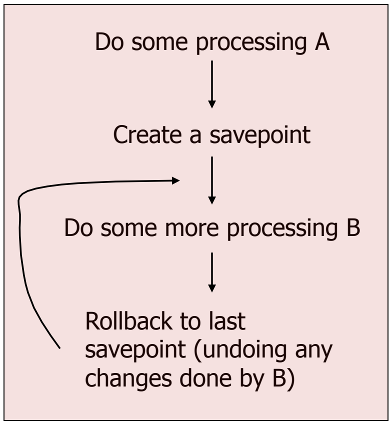
72

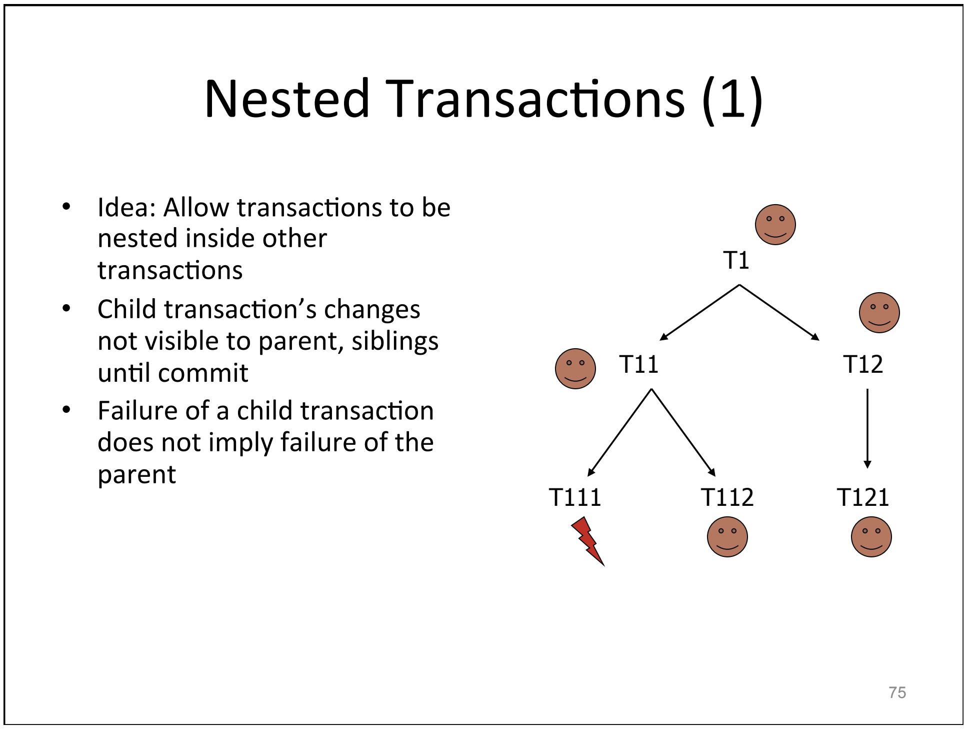
EXTENDED TRANSACTION MODELS

Savepoints

- Idea: allow the state of a transaction to be rolled back to a certain point in execution
- Not necessary to roll back the entire transaction

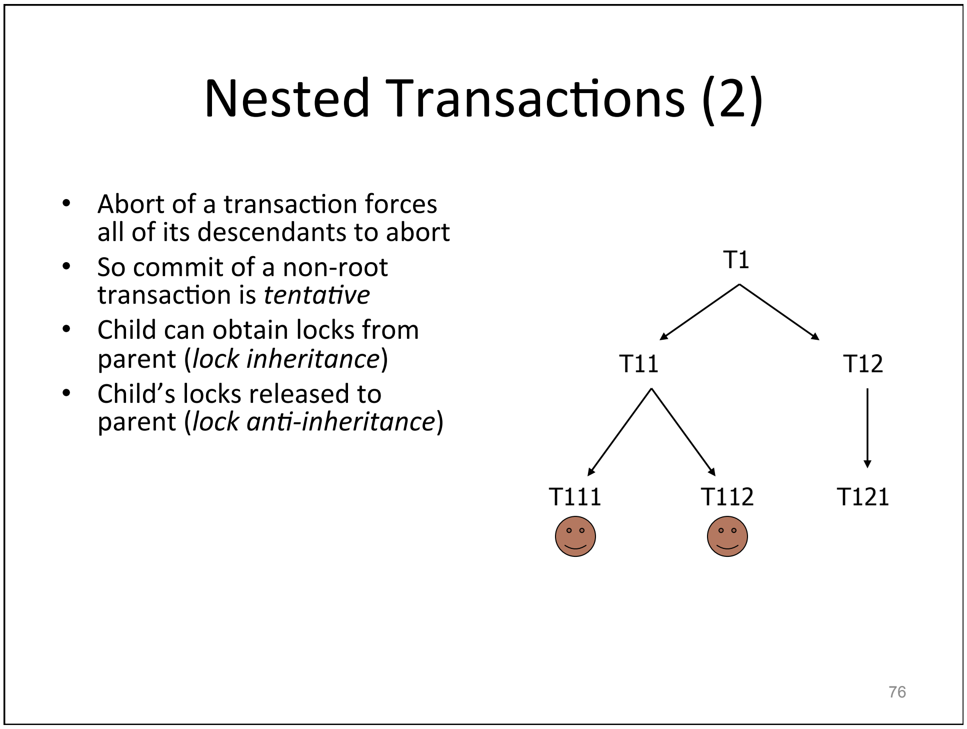
Transaction Execution


Do some processing A
Create a savepoint
Do some more processing B
Rollback to last savepoint (undoing any changes done by B)



Nested Transactions (1)

- Idea: Allow transactions to be nested inside other transactions
- Child transaction's changes not visible to parent, siblings until commit
- Failure of a child transaction does not imply failure of the parent

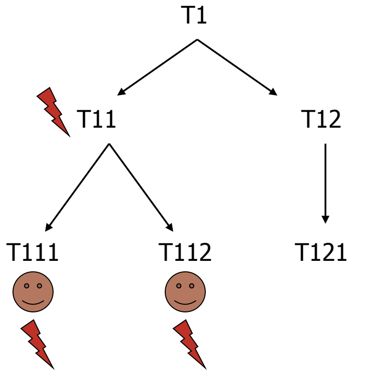


Nested Transactions (2)

- Abort of a transaction forces all of its descendants to abort
- So commit of a non-root transaction is *tentative*
- Child can obtain locks from parent (*lock inheritance*)
- Child's locks released to parent (*lock anti-inheritance*)

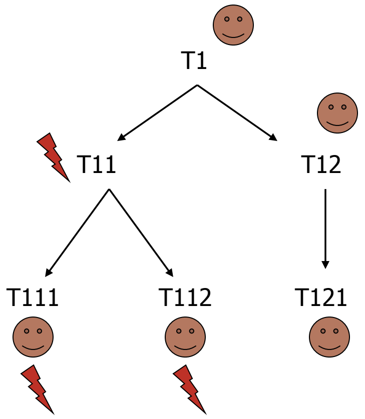
Nested Transactions (2)

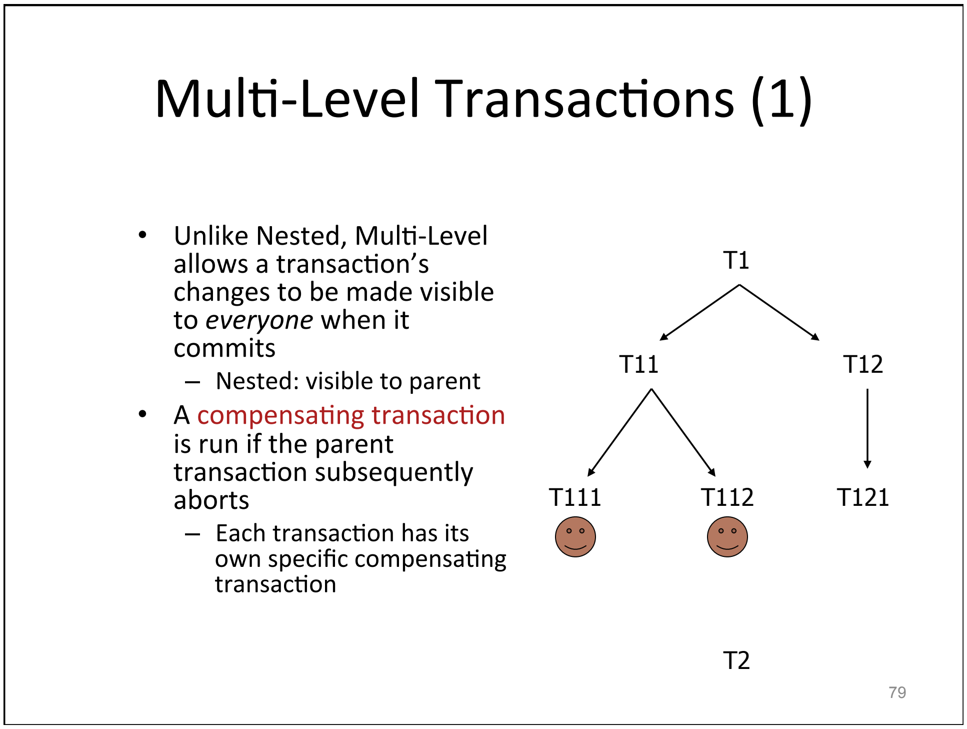
- Abort of a transaction forces all of its descendants to abort
- So commit of a non-root transaction is *tentative*
- Child can obtain locks from parent (*lock inheritance*)
- Child's locks released to parent (*lock anti-inheritance*)



Nested Transactions (2)

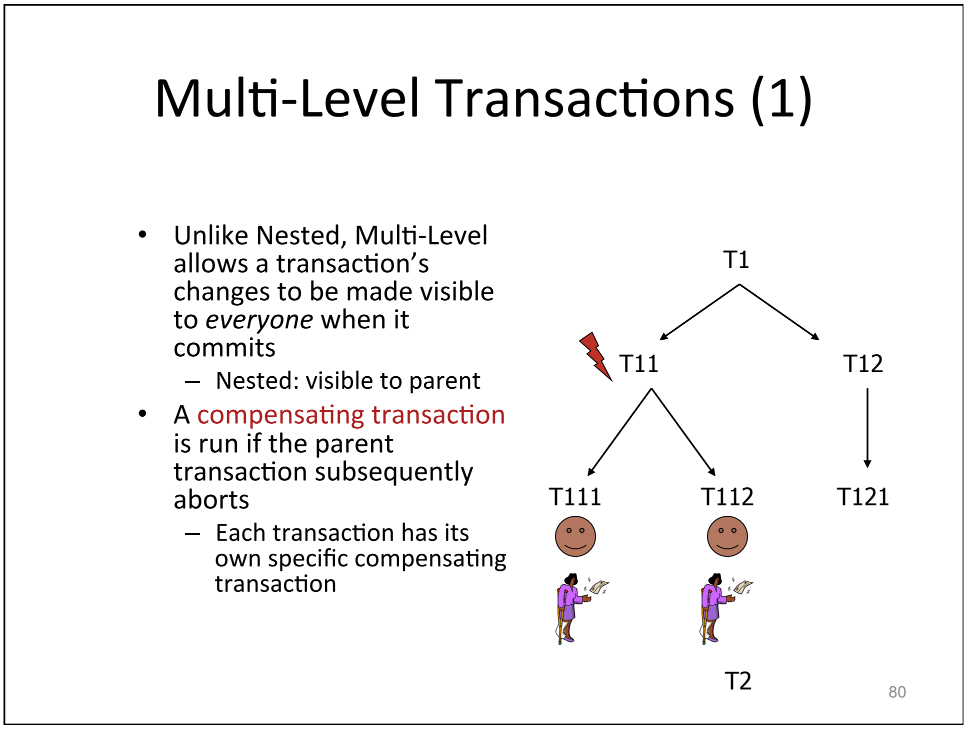
- Abort of a transaction forces all of its descendants to abort
- So commit of a non-root transaction is *tentative*
- Child can obtain locks from parent (*lock inheritance*)
- Child's locks released to parent (*lock anti-inheritance*)





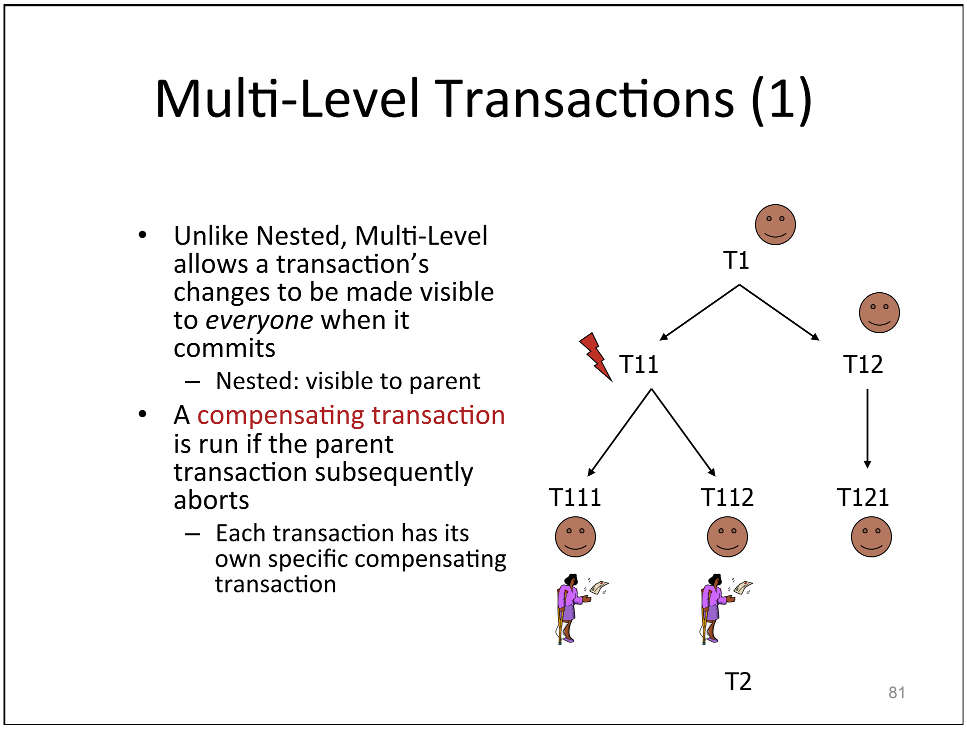
Multi-Level Transactions (1)

- Unlike Nested, Multi-Level allows a transaction's changes to be made visible to *everyone* when it commits
  - Nested: visible to parent
- A compensating transaction is run if the parent transaction subsequently aborts
  - Each transaction has its own specific compensating transaction



Multi-Level Transactions (1)

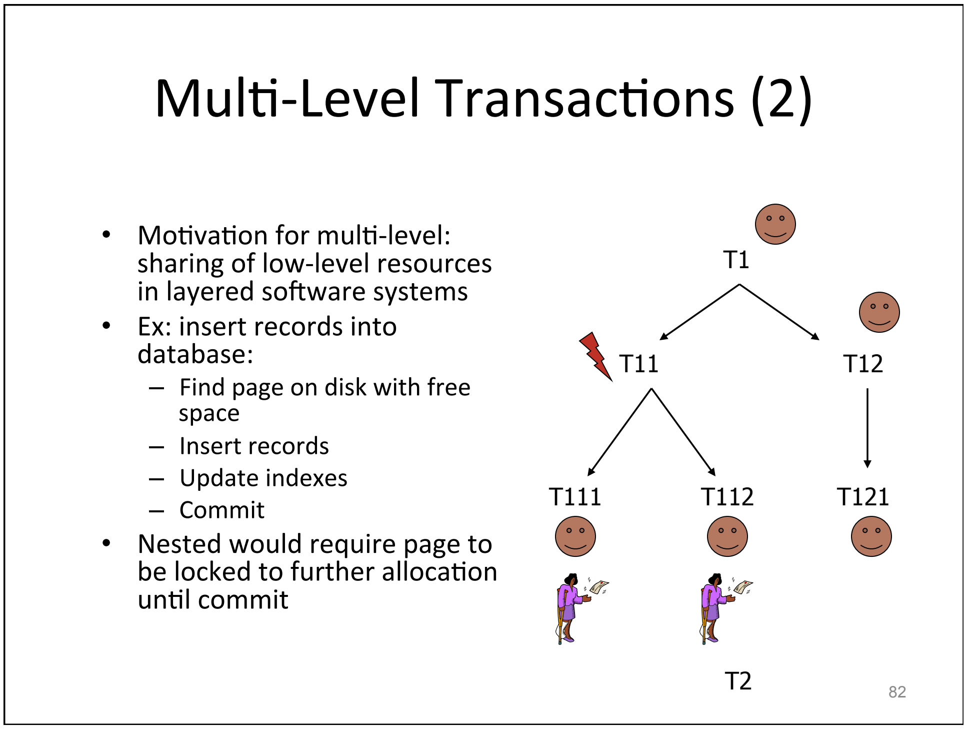
- Unlike Nested, Multi-Level allows a transaction's changes to be made visible to *everyone* when it commits
  - Nested: visible to parent
- A compensating transaction is run if the parent transaction subsequently aborts
  - Each transaction has its own specific compensating transaction



Multi-Level Transactions (1)

- Unlike Nested, Multi-Level allows a transaction’s changes to be made visible to *everyone* when it commits
  - Nested: visible to parent
- A compensating transaction is run if the parent transaction subsequently aborts
  - Each transaction has its own specific compensating transaction

T1
T11
T12
T111
T112
T121
T2



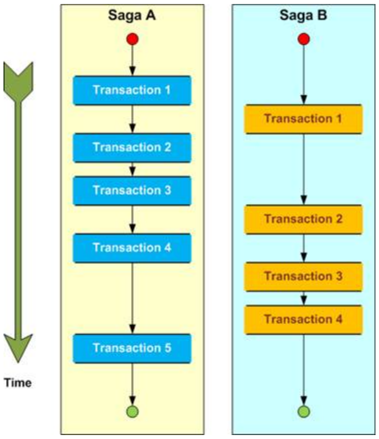
Multi-Level Transactions (2)

- Motivation for multi-level: sharing of low-level resources in layered software systems
- Ex: insert records into database:
  - Find page on disk with free space
  - Insert records
  - Update indexes
  - Commit
- Nested would require page to be locked to further allocation until commit

T1
T11
T12
T111
T112
T121
T2

Sagas (1)

- Transactions are incompatible with long-lived applications
  - Locks must be held to ensure ACID properties
- Sagas relax ACID properties, allow intermediate states to be visible to outside
- A saga is a sequence of transactions
  - "long-lived transaction"

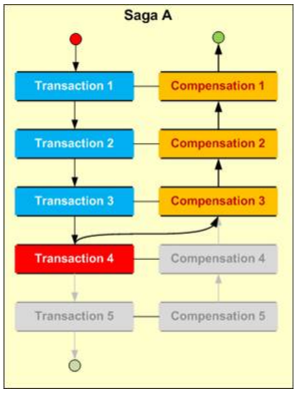


Sagas (2)

- Each transaction T_i has a corresponding compensating transaction CT_i
- If T_k aborts in a transaction, then the compensations

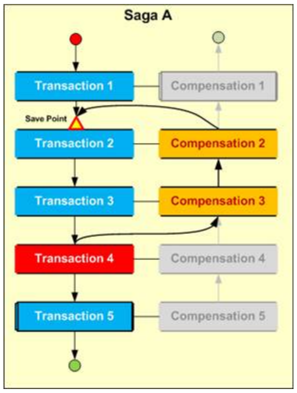
  $CT_{k-1},\ldots,CT_2,CT_1$

  are executed, in that order
  - Explicit rollback



Sagas (3)

- Sagas also support forward recovery using savepoints
- If state is persisted between transactions, then compensating transactions roll back to that savepoint
- Saga continues from that point forward


Saga A
Transaction 1
Compensation 1
Save Point
Transaction 2
Compensation 2
Transaction 3
Compensation 3
Transaction 4
Compensation 4
Transaction 5
Compensation 5