PURDUE
UNIVERSITY

CS54200: Distributed Database Systems

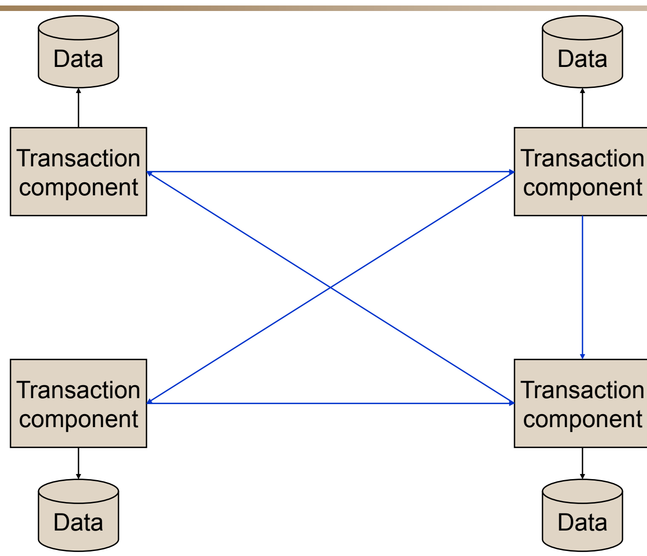
Distributed Recovery

6 February 2009

Prof. Chris Clifton

Indiana
Center for
Database
Systems

What is a Distributed Transaction?



Distributed Recovery

- Assume a distributed architecture composed of several local TM, Schedulers, RM, and CMs .
- Ignore replication for now.
- Allow each site's CM to manage the local cache for local data.
- Allow each site's RM to manage recovery (as in the centralized case) using any of the recovery algorithms discussed earlier.
- Will this work, as with distributed concurrency control?

5

NO!

- What is the key difference?
- Atomicity – global agreement.
- Isn't *global agreement* required for CC as well?
- CC is handled locally, without communication.
- Global agreement is required for CC:
  - 2PL – avoiding deadlocks; but this is avoided by strict 2PL.
  - TO – to ensure agreement on relative ordering of txns; this is piggy-backed with the request message sent by TM to schedulers.

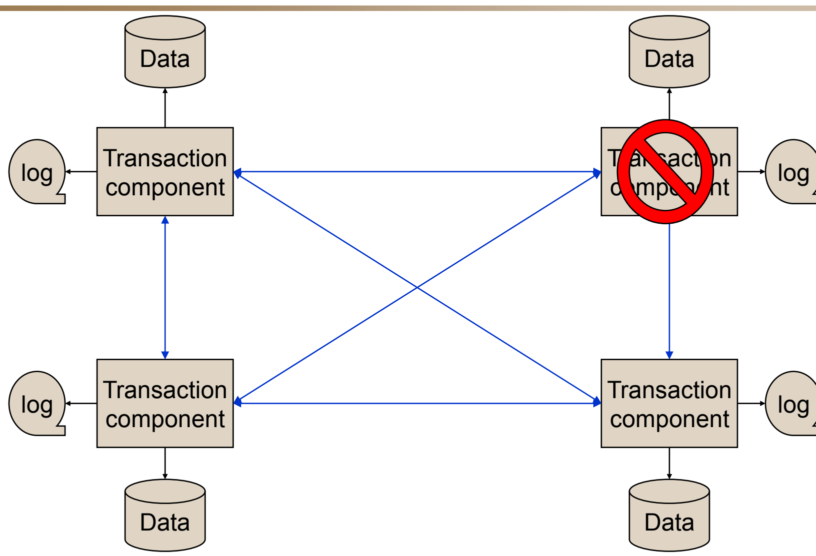
6

Atomic Commit Protocol

- Distributed recovery requires global agreement – every site must agree whether to commit or abort a txn – this must be an atomic action.
- Protocols that ensure a consistent decision across distributed sites are called Atomic Commitment Protocols (*ACP*s).
- Where does the problem come from?
- Failures!!!

7

Committing a Distributed Transaction



Failures

- Site failures:
  - Assume non-Byzantine failures
  - Fail-stop
  - Partial or total
- In the absence of failures, each pair of sites can communicate.
- Communication failures:
  - When two sites, neither of which has failed, are unable to communicate (via any route)
  - May lead to partitioning.

9

Failures

- If a message is undeliverable, we assume that it is dropped.
- Failures are detected using timeouts.
- Can lead to *false detections*, as well as *delayed detections*.
- Must make a judicious choice of the timeout interval.
- *A failed site may recover at any time* – will need to execute the recovery process at that time.

10

ACP

- The steps in an ACP are as follows:
  - TM gets a commit operation from the txn.
  - ACP needs to arrive at a single, consistent decision to commit or abort based upon the state of the txn at each site i.e.
    - Scheduler
    - DM (ensure that redo rule is satisfied) if there were only read operations at a site, ACP doesn't need to consult DM
  - Can do this by polling all sites.
  - Send the decision to each site.

11

System Model

- We abstract the problem to one of reaching a decision between distributed processes.
- There are two types of processes:
  - *Coordinator:* This is the site which initiates the ACP – I.e. where the TM gets the commit operation. Note that the decision if txn want to abort is easy to arrive at.
  - *Participants:* all other processes involved.

12

System Model

- Initially, the coordinator knows all the participants, but the participants don't know each other.
- Assume that each site has a distinct log called the Distributed Transaction Log (*DT Log*).
- Each process can vote *yes* or *no* .
- Each process can reach a decision : *commit* or *abort*.

13

ACP Requirements

- AC1: All processes that reach a decision reach the same one.
- AC2: A process cannot reverse its decision after it has reached one.
- AC3: The *Commit* decision can only be reached if *all* processes voted *Yes*.
- AC4: If there are no failures and all processes voted *yes*, then the decision will be to commit.
- AC5: Consider any execution containing only failures that the ACP is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision.

14

ACP terminology

- The period between sending a *yes* vote and reaching a decision is called the *uncertainty period*.
- When a process must await the repair of failures before proceeding, we say that it is *blocked*. E.g. when a failure disables comm. between a process and all other sites when the process is uncertain.
- If a process fails while uncertain, it cannot reach a decision on its own upon recovery – it must communicate with other processes. An ACP that avoids such situations has the *independent recovery* property.

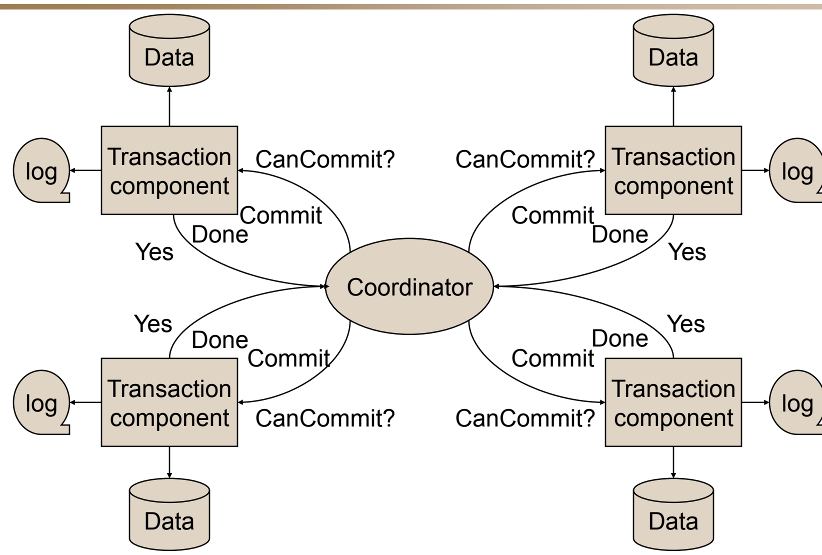
15

2 Phase Commit Protocol (Lamport '76, Gray '79)

1. *Coord* sends *VOTE_REQ* to all *participants*.
2. Each *P* sends a msg back with its vote: *YES* or *NO*. If it votes *NO*, it decides *ABORT* and stops.
3. The Coord collects all votes.
   - If all are *YES* and its own vote is *YES*, it decides *COMMIT* and sends *COMMIT* msgs to each participant. Stop
   - Otherwise, it decides *ABORT* and send *ABORT* msgs to all participants that voted *YES*. Stop.
4. Each participant that voted *YES* waits for the coord's decision, decides accordingly and stops.

19

Two-Phase Commit



Complications

- If no failures take place this ACP works fine.
- However, if there are failures, we need to specify what happens when:
  – There is a timeout while waiting for a message; or
  – A site crashes and then recovers during the ACP?
- Timeout actions:
  – Participant waiting for a *VOTE_REQ*: unilaterally abort.
  – Coord waiting for a vote: decide *ABORT* and send msg to all sites that voted *yes*.

22

Timeout Actions

- While waiting for the decision: tricky – use a *termination protocol*
- Suppose that participant *P* needs to determine the decision
  - *P* can wait until it can communicate with Coord
  - This is simple, but could unnecessarily block *P*
  - Instead, *P* could learn of the decision from some other participant.
- The second option can be used with the *cooperative termination protocol*

23

Cooperative Termination Protocol

- Process *P* sends a *decision_REQ* message to every participant, *Q. P* learns of the other participants from the *VOTE_REQ* message sent by the Coord.
- *Q* does the following:
  - If *Q* has already decided, then it send its decision to *P*
  - If Q has not yet voted, then it can unilaterally abort and send *ABORT* to P.
  - If Q is also uncertain then it cannot help *P* – both are blocked.

24

Handling site failure in 2PC

- We use a distributed transaction log to record necessary information about termination protocols, in order to recover correctly.
- The DT log can be a part of the regular log too.
- It works as follows:
  - When *Coord* sends a *VOTE_REQ*, it writes a *start-2PC* record (before or after sending message).
  - If a participant votes *yes*, it writes a *yes* record before sending the vote. This record contains the identities of the coordinator and other participants (as given by the initial message of the coord).

25

DT Log

- If the participant votes *no,* it writes an *abort* record, either before or after sending the vote.
- Before the *Coord* sends a commit decision, it writes a *commit* record.
- When the *Coord* sends abort, it writes the *abort* record to the log
- After receiving commit(abort), a participant writes a *commit(abort)* record to its log.

26

Recovery

- When a site recovers, the fate of a distributed txn is determined as follows.
- If the DT log contains a *start-2PC* record, then the recovering site, *s*, was the coordinator
  - if it also contains a *commit* or *abort* record, then the coord had reached a decision before failure.
  - if neither is found, the coord can now unilaterally decide ABORT.
- If the DT log doesn't contain the *start-2PC* record, then the site was a participant. There are three cases:

27

Recovery (contd.)

- The DT log contains a *commit* or *abort* record – I.e. participant had reached a decision.
- The DT log does not contain a *yes* record: either the participant failed before voting, or voted *NO*. It can therefore unilaterally decide to *ABORT*.
- The DT log contains a *yes* record, but no *commit* or *abort* record: participant failed during the uncertainty period – use the termination protocol to determine fate.

28

Garbage Collection

- As with the regular log, the DT log needs to be garbage collected. There are two basic rules:
- GC1: A site cannot delete entries of txn *T* from the DT log at least until its RM has processed *RM-Commit* or *RM-Abort.*
- GC2: At least one site must not delete the records of txn *T* from its DT log until that site has received messages indicating that *RM-Commit(T)* or *RM-Abort(T)* has been processed at all other sites where *T* executed. (can be done by the coordinator).

29

How good is 2PC?

- *Resiliency*: what types of failures does it tolerate?
  - *Site* and *communication* failures (even partitioning)
- *Blocking*: does it block, if so when?
  - Yes. If a process times out in its uncertainty period and can only communicate with other uncertain processes.
- *Time Complexity*: How many rounds?
  - With no failures: 3 rounds are needed.
  - With failures, we need a termination protocol, that could add 2 more rounds.
  - Each failure could result in these extra rounds,but they could overlap, so we count only 2 rounds.

30

Message Complexity

- *Message Complexity*: with n participants
  - With no failures: $3n$ messages.
  - If there are m sites that invoke the termination protocol, then mn *DECISION_REQ* messages are sent, at most $(n-m+1)$ could respond. With each round of the termination protocol, one less process is in its uncertainty period, and thus one more could respond, therefore the maximum number is:

$$mn + \sum_{i=1}^{m}(n - m + i) = 2nk - \frac{m^2}{2} + \frac{m}{2}$$

which is at most $n(3n+1)/2 + 3n$ in total.

31

Alternative 2PC

- Decentralised: complete graph. Better time complexity.
- Linear: better message complexity.

	Centralised	Decentralised	Linear
Time	3 rounds	2 rounds	$2n$ rounds
MSG	$3n$	$n+n^2$	$2n$

32

Two-Phase Commit: Problems

- Blocks on failure
  - Timeout before abort if participant fails
  - *All participants must wait for recovery if coordinator fails*
- While blocked, transaction must remain Isolated
  - Hold locks on data items touched
  - Prevents other transactions from completing

Two-Phase Commit (Lamport '76, Gray '79)

- Central coordinator initiates protocol
  - Phase 1:
    - Coordinator asks if participants can commit
    - Participants respond yes/no
  - Phase 2:
    - If all votes yes, coordinator sends Commit
    - Participants respond when done
- Blocks on failure
  - Participants must replace coordinator
  - *If participant and coordinator fail, wait for recovery*
- While blocked, transaction must remain Isolated
  - Prevents other transactions from completing

Negative Results

- Total failures + lack of independent recovery give rise to blocking.
- Can we eliminate uncertainty periods?
- Unfortunately, there are two well-known results:

1. *There are no ACPs that eliminate blocking if communication or total failures are possible.*
2. *No ACP can guarantee independent recovery of failed processes.*

35

Formal Recovery Models (Skeen & Stonebraker '83)

- Formal models for commit protocols
  - Transaction state
  - Failure
- Protocol types
  - Commit
  - Termination
  - Recovery
- Necessary Conditions for non-blocking
  - Leads to Three-Phase Protocol

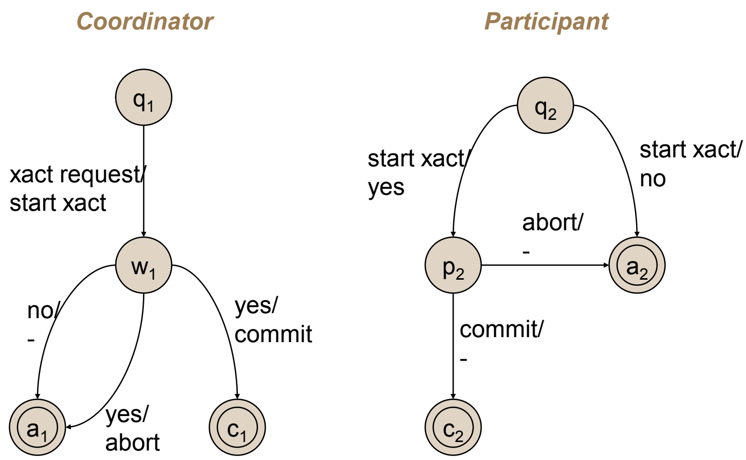
Background

- Transaction has "commit point"
  - Failure after ⇒ transaction visible
  - Failure before ⇒ abort as part of recovery
- Protocol needed to ensure commit/abort decision unanimous
  - Any site can abort before first site commits
  - No site can abort after first site commits
- Non-Blocking Protocol: Failure doesn't cause operational sites to suspend

Transaction Model

- Network:
  - Point to point communication
  - Maximum delay *T* or timeout
    - If timeout, sender can assume recipient of network failure
    - If network failure, sender doesn't know if message received
- Each site can be viewed as Finite State Automaton
  - Sites have three state "classes":
    - Initial: Allowed to abort the transaction
    - Abort: Can't transition to non-abort
    - Commit: Can't transition to non-commit
  - Nondeterministic with respect to protocol, i.e., State transitions may occur independent of protocol due to local actions
  - State diagram is acyclic (assures termination)
  - Site transitions asynchronous
  - State transitions atomic

State Transition Graph for Two-Phase Commit

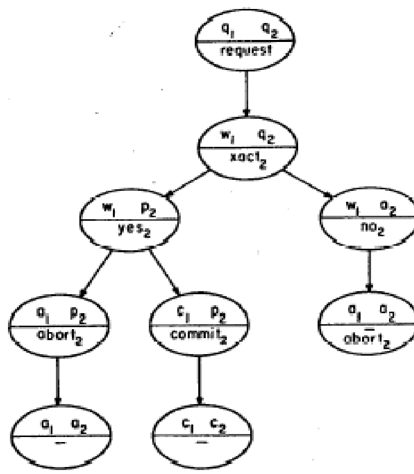


Transaction Model

- Global Transaction State
  - Vector of local states
  - Outstanding messages in the network
  - Final if all local states in final state
  - Inconsistent if vector contains both commit and abort states
  - Global state transition occurs when local state transition occurs
- Reachable State Graph
  - Possible global transitions
  - Protocol correct if and only if reachable state graph has:
    - No inconsistent states
    - All terminal states are final states
- Local states potentially concurrent if a reachable global state contains both local states
  - Concurrency set C(*s*) is all states potentially concurrent with *s*
- Sender set S(*s*) = {local states *t* | *t* sends *m* and *s* can receive *m*}

Reachable Global State Graph for Two-Phase Commit

- Leaf nodes are terminal states
  - All contain only final states
- No nodes have both "abort" and "commit"
  - Protocol consistent
- Therefore 2-phase commit is operationally correct



Failure Models

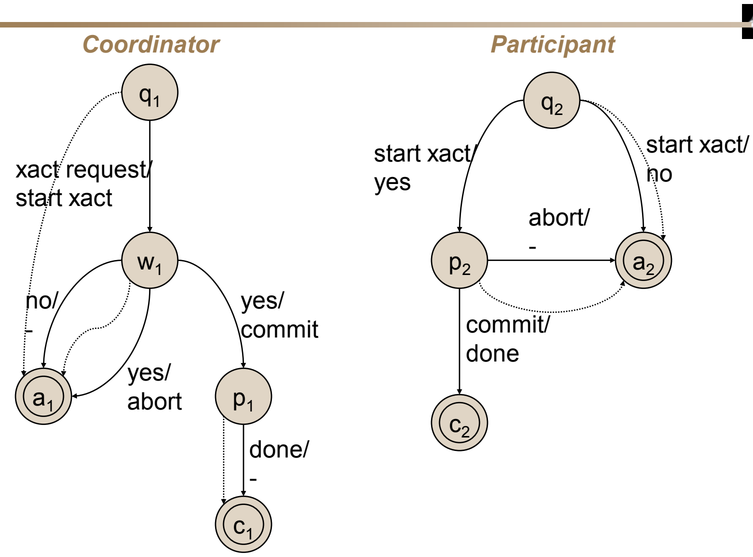
- Site failure assumed when expected message not received in time
  - Modeled as "failure transition"
    Assumption: A site knows it has failed
  - Allow multiple failure transitions from a state
    Different types of failure
- Independent Recovery
  - Transition directly to final state without communication
  - Paper discusses only two site case

Single Site Failure

- Lemma 1: If a protocol contains a local state s with both abort and commit in C(s), cannot independently recover
  - C(s) has both abort and commit
  - s cannot fail to either abort or commit

  2PC: C(p_2) has both commit and abort!
- Rule 1: If C(s) has commit, insert failure transition from s to commit, else insert failure from s to abort

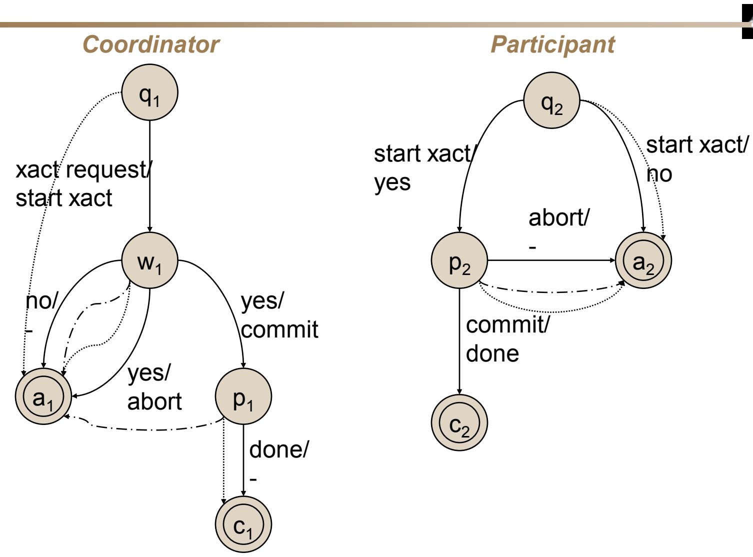
Modified 2PC with Rule 1



Handling Timeout

- Rule 2: For each intermediate state s
  - if t is in S(s) and t had a failure transition to a commit (abort) state, then
  - assign a timeout transition from s to a commit (abort) state
- Assumption: Failed state will independently recover
  - Rule 1 forces transition to commit / abort
  - Rule 2 forces "live" transaction to do same

Modified 2PC with Rules 1,2



Theorem: Sufficient Conditions for Handling Failure

- Rules 1 and 2 are sufficient for designing protocols resilient to a single site failure
- Proof: Let P be protocol s.t. there is no s where C(s) contains commit and abort
  - P' is P modified by Rules 1 and 2
  - Site 1 fails in state s_1 when Site 2 in s_2
    - Transitions to f_1 inconsistent with f_2
- Case 1: Site t_2 in final state f_2
  - Implies f_2 in $C(s_1)$ – violates Rule 1
- Case 2: Site 2 in nonfinal state, timeouts to f_2
  - Implies $s_1 \in S(s_2)$ – violates Rule 2

Two site failure

- Theorem: No protocol using independent recovery resilient to arbitrary two site failures
  - Holds only if failures concurrent (both sites fail without knowing other has failed)
- Proof: Assume path in global state graph $G_0, \ldots, G_m$, all sites recover to abort from G_0, to commit from G_m
- Let G_k be first state where first site j recovers to commit
  - j recovers to abort in G_{k-1}
  - j was only site to transition between G_k and G_{k-1}
  - All other sites will recover same in G_k and G_{k-1}
  - So either G_k or G_{k-1} inconsistent if j and another site fail
- Key: No non-blocking *recovery*
  - Doesn't mean operational sites have to block

CS54200: Distributed Database Systems

Three-Phase Commit

6 February 2009

Prof. Chris Clifton

Indiana Center for Database Systems

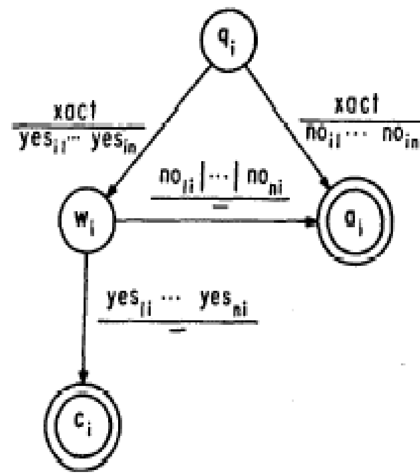
Introduction

- No ACP can eliminate blocking if total failures or total site failures are possible.
- 2PC may cause blocking even if there is a non-total site failure – how?
- Introduce a new ACP which eliminates blocking in the absence of comm. and total site failures.
- Unfortunately, it does not tolerate all communication failures – some cases may result in inconsistent decisions.
- A variation of this protocol avoids these problems.

50

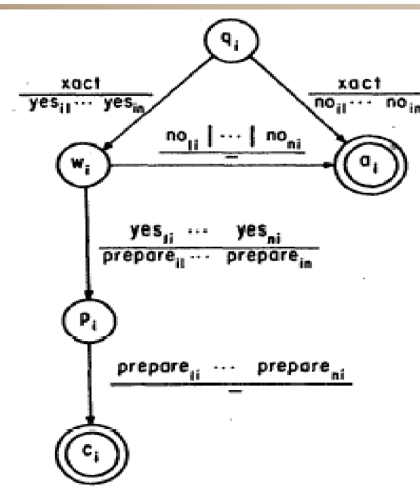
Centralized vs. Decentralized Protocols

- What if we don't want a coordinator?
- Decentralized:
  - Each site broadcasts at each round
  - Transition based on all messages received
- Decentralized Two-Phase Commit →



Decentralized 3-Phase Commit (Skeen '81)

- Send start message
  - When ready, send yes/no
- If Any no's received, abort
- If all yes's, send prepare
  - Failure → commit
  - Timeout → abort
- When prepares received, commit



3 Phase Commit

- Assume no communications failures ➔
  - every pair of operational sites can communicate
  - A time-out implies that the sender is down (I.e. it is not doing anything).
- 2PC causes blocking because uncertain operational sites cannot be sure that a site didn't commit before failure.
- We want to have the following property:
  - *NB: If any operational process is uncertain then no process (operational or failed) can have decided to Commit.*
- 3PC is designed to satisfy *NB*.

53

3PC

- The problem with 2PC is that the coordinator sends *Commit* messages while the participants are uncertain.
- Thus participants can *decide* commit while some other participants are uncertain.
- 3PC avoids this by sending *pre-Commit* messages instead of *Commit* messages, thereby moving every participant out of the uncertainty period before any participant commits.
- After coord receives ack for *pre-Commits*, it sends commit, allowing participants to commit.

54

3PC Steps

1. Coord send *VOTE_REQ* messages
2. Participants respond with vote. If it sends *NO*, it *decides* *Abort* and stops.
3. The coord collects all votes and determines whether to commit or abort. If it *decides* *Abort* it sends the *Abort* to all sites. Otherwise, it sends *pre-Commit* messages to all participants.
4. A part that voted *YES* waits for a *pre-Commit* or *Abort* message. If it receives a *pre-Commit*, it sends an *ACK* to the Coord, otherwise it *Aborts* and stops.

55

3PC Steps (contd.)

5. The Coord collects all *ACK*s. When they have been received, it *decides* Commit, and send *Commit* messages to all participants, and stops.

6. A participant waits for a *commit* from the Coord.

When it receives that message, it *decides* *Commit* and stops.

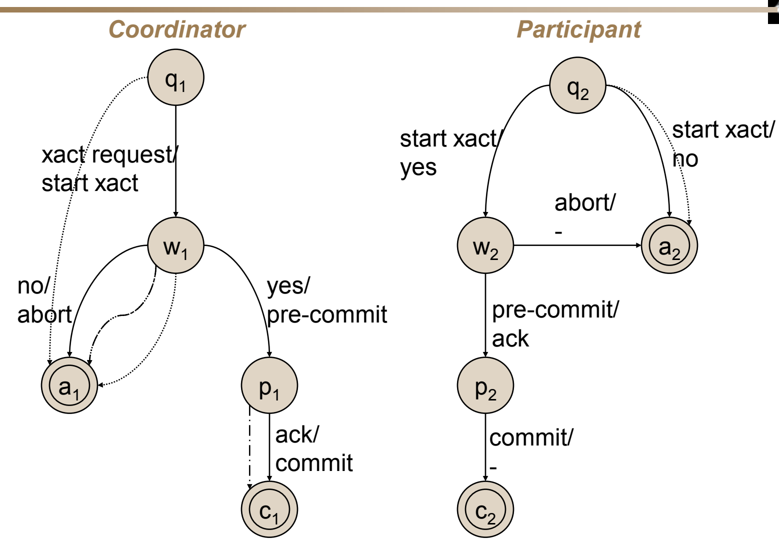
What are the actions for timeouts and recovery?

56

What about non-independent recovery?

- Previous protocols assume independent recovery
  - Always know proper decision when recovering from failure
  - Problem: Operational processes block with multiple failures
- Solution: Recovery may need to request help

3PC assuming timeout on *receipt* of message



Timeout Actions

- Steps 1 and 3: unilaterally <u>*decide*</u> *ABORT*
- Step 5: A participant has failed. May have recvd the *pre-Commit* before failure. Coordinator ignores the failure (participant must be willing to commit) even though some *failed* site may be *uncertain*.
- Steps 4 and 6: can't decide unilaterally. Step 4 is as before. Why is step 6 difficult? Only a *commit* can be received since *pre-Commit* has been received. Why can't the participant ignore the timeout and decide commit?

59

Timeout Actions

- It can't decide because some participants may still be uncertain. This would violate *NB.*
- At this point, the participant should determine if any of the operational sites are still uncertain – use a *termination protocol*.
- A process can be in one of the following states:
  - *Aborted*;
  - *Uncertain*;
  - *Commitable*; or
  - *Commited*.

60

Solution: Termination Protocol

- If participant times out in w_2 or p_2:
  - Elect new Coordinator

    If coordinator alive, would have committed/aborted
- New coordinator requests state of all processes. Termination rules:
  - If any aborted, broadcast abort
  - If any committed, broadcast commit
  - If all w_2, broadcast abort
  - If any p_2, send pre-commit and enter state p_1

Termination Protocol

1. Upon timeout, the participant initiates a *leader election*, involving all operational sites.
2. The new coordinator sends a *STATE_REQ* message to all processes that participated in election.
3. The Coord uses the *termination rule:*

TR1: If some process is Aborted, the coord decides *Abort*, sends *ABORT* messages, and stops.

TR2: If some process is Committed, the coord decides *Commit*, sends *COMMIT,* and stops.

62

Termination Protocol (contd.)

TR3: If all processes report that they are uncertain, the Coord decides *Abort,* sends *ABORT* messages, and stops.

TR4: If some process is commitable but none is commited, send *PRE_COMMIT* messages to all uncertain processes; wait for *ACK*s. Upon receiving these, decide *Commit*, send *COMMIT* messages and stops.

63

3PC Termination

- What happens if we get failures during the termination protocol?
- The Coord will ignore failed participants.
- If the Coord fails, then a new one is elected. This can go on until all sites have failed – total failure!
- Note that a site that recovers during the termination protocol is not allowed to take part.
- Such processes use the recovery operations.

64

Recovery Actions

- A recovering participant, *p,* first determines its state with respect to the transaction. If it failed
  - Before sending *YES*, unilaterally abort.
  - After receiving *Commit/Abort*, decided.
  - Otherwise, it must communicate with other processes.
- Since there is no blocking, a decision has already been made, or is being made, or *p* is the first process to recover from a total failure.
- In the first 2 cases, it will eventually get the decision.
- Note that even if *p* had recvd a *pre-Commit*, it cannot decide to *COMMIT.* Decision may be to abort due to termination protocol!

65

Terminates

- Lemma: Only one of termination rules can apply
- Theorem: In the absence of total failures, 3PC with termination protocol does not block
  - If coordinator alive, terminates after timeout. Otherwise elect new coordinator.
    - By Lemma, one of rules selected → decision
  - If new coordinator fails, repeat
    - Either succeeds, or all processes failed

Theorem: All operational processes agree

- No failure: all messages sent to each process, so each agree
- Induction: works for *k* failures. On *k+1*:
  - First rule: *p* has aborted. So before failure, *p* didn't vote or voted no, or received abort.
    - No process could have previously committed
  - Second: *p* committed. So before failure, *p* had received commit
  - Third: Will abort. No previous commit. Since all operational in w_2, no process could be committed
  - Fourth: Will commit. Assume *p* previously aborted – no process could have entered p_2

What about failed processes?

- Preceding assumes failed processes stay failed
  - We've removed failure transitions for independent recovery
- Solution: Recovering site requests state from operational sites
  - Since 3PC non-blocking, will eventually get response from operational site
  - Same process for recovery from w_2 or p_2

What if all sites fail?

- If not in w_2 or p_2, recover independently
- If last site to fail, run termination protocol
  - Only need to run with self
    - *Would have been okay before failure*
  - Thus independent recovery
- Otherwise ask other sites when they recover

Total Failures

- Upon recovery from a total failure, a process is blocked unless:
  - It was the last failed site
  - It had decided before failure.
- If it was the last site to fail, it invokes the termination protocol as the leader.
- All processes that have recovered by this time can participate.
- Note that the inclusion of the last failed site in such a situation is critical.

70

Election Protocol

- Order the sites.
- All sites exchange their *ID*s during election.
- The site with the lowest *ID* is the leader.
- Each site maintains a list of processes that it believes are operational, $UP_{p.}$
- When a site detects the failure of the coord, it removes it from Up_p and considers the process with smallest *ID* in Up_p to be the new leader. If it is not the new leader, it sends a *UR_ELECTED* message to the leader.

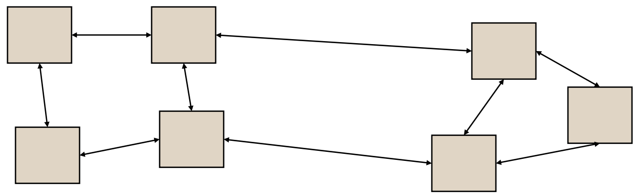
71

Election Protocol (contd.)

- A site that receives *UR_ELECTED* leads
- A site that receives a *STATE_REQ* from a new leader ignores it if the *ID* is lower that what it considers to be the current leader, or
- Assumes it to be the new leader, removes all entries with smaller values from $UP_{p.}$
- The Up_p lists are useful for determining the last failed site too.
- A set of recovered sites *R*, contains the last failed site if $R \supseteq \bigcap_{p\in} UP_p$

72

Communication Failures



- Problem: Network partition indistinguishable from process failure
- Solution: Need responses from majority
  - Not non-blocking
  - But non-blocking not possible!
- More difficult when transient partition
  - Election of multiple coordinators with majority

Evaluation of 3PC

- *Resiliency and blocking*: site failures only. Non-blocking unless total failure.
- *Time Complexity*:
  - With no failures: 5 rounds.
  - With failures: Each invocation of termination protocol adds 5 more rounds + UR_ELECTED. Thus with *f* failures, 6*f* + 5 rounds.

74

Evaluation of 3PC

- *Message Complexity*:
  - With no failures: *5n* rounds
  - With failures: each round of termn. Protocol the number of message is the number of remaining part.
  - In *i*-th invocation, there are at most *(n-i)* left, so the number of messages is at most *6(n-i).*
  - With *f* failures, at most

$$5n + \sum_{i=}^{f} \quad n - \quad) = \quad (f + \quad)(2n - \quad) - \quad .$$

75