DoC 437 - 2009

Distributed Algorithms

Part 3: Atomic Commitment

What is atomic commitment?

- Motivation

  when transactions update data in a distributed system, *partial failures* can lead to *inconsistent results*

- Atomic commitment protocol

  ensures that a distributed transaction terminates *consistently* at all participating sites even in the *presence of failures*

What is a distributed transaction?

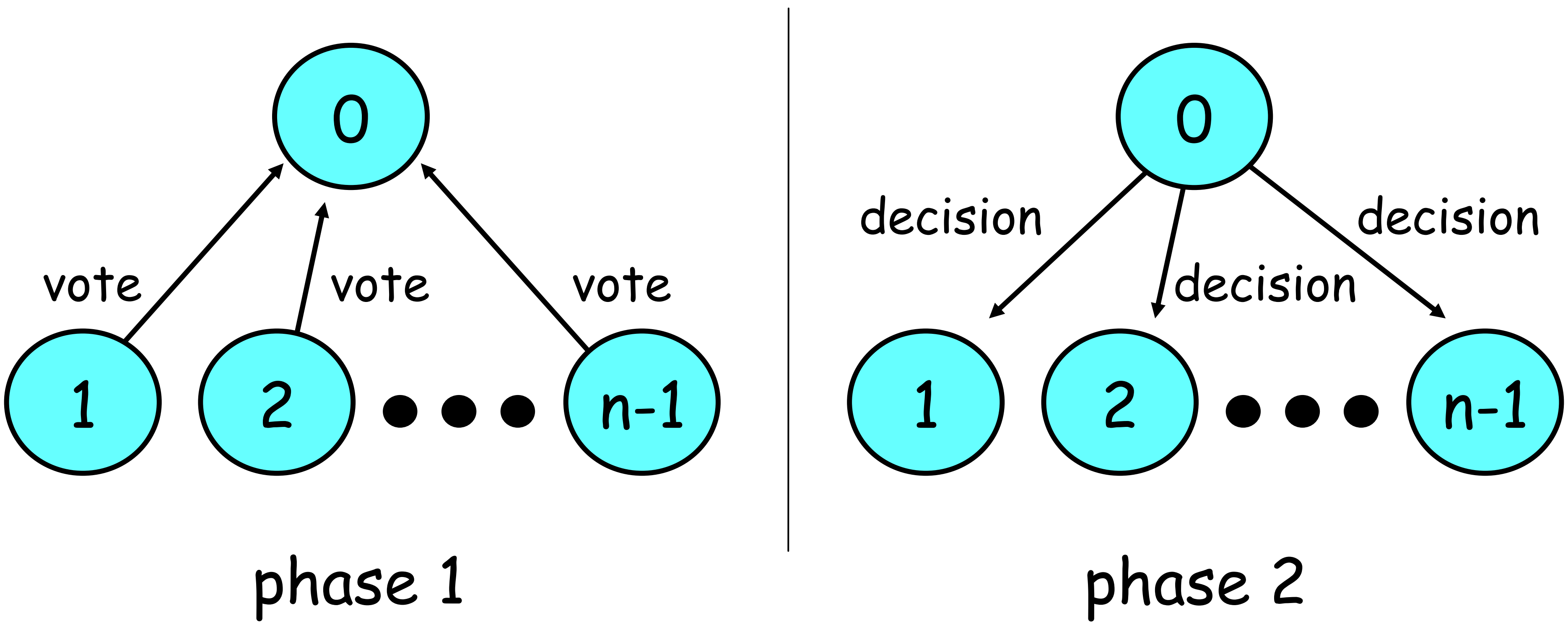
- The execution of a program that accesses data at multiple sites
- In the absence of failures, execution brings data from one consistent state to another
- Failure atomicity

  preserving data consistency in the presence of failures

  a concern orthogonal to concurrency control

Familiar solution

2PC: two-phase atomic commitment protocol



Two-phase commit

the basic model

- Three kinds of processes

  participants: the set of processes that perform updates on behalf of a transaction

  invoker: a single process that distributes the transaction to the participants

  coordinator: a single process that orchestrates the conclusion of a transaction among the participants

- For simplicity, consider invoker and coordinator as participants

Two-phase commit

the basic model

- After performing their local actions, each participant selects between two possible “opinion“ values

  yes: willing and able to make changes permanent

  no: unwilling or unable to make changes permanent

- Participants engage in a “centralized” coordination step to decide the outcome of the transaction

Two-phase commit

the basic model

- A coordinator is chosen, satisfying the following axioms

  AX1: at most one participant will assume the role of coordinator

  AX2: if no failures occur, one participant will assume the role of coordinator

  AX3: constant Δ_c such that no process assumes the role of coordinator more than Δ_c time units from the start of the transaction

Two-phase commit
the basic model

- Synchrony

    synchronous

- Communication

    message-passing network

        point-to-point and broadcast

    assumed to be failure free

    channel delay bounded by δ time units

Two-phase commit

the basic model

- Failure model for processes

  a process is either *operational* or *down*

  failures cause an operational process to go down

    this state change is called a *crash*

  a down process may execute a recovery process to become operational again

  a process is *correct* if it has never crashed, otherwise it is *faulty*

- Timeout can be used to detect process failures

Two-phase commit

the desired outcome

- Eventually, each participant must select between two possible and irreversible “decision” values, and all participants must agree

  commit: all participants make updates permanent

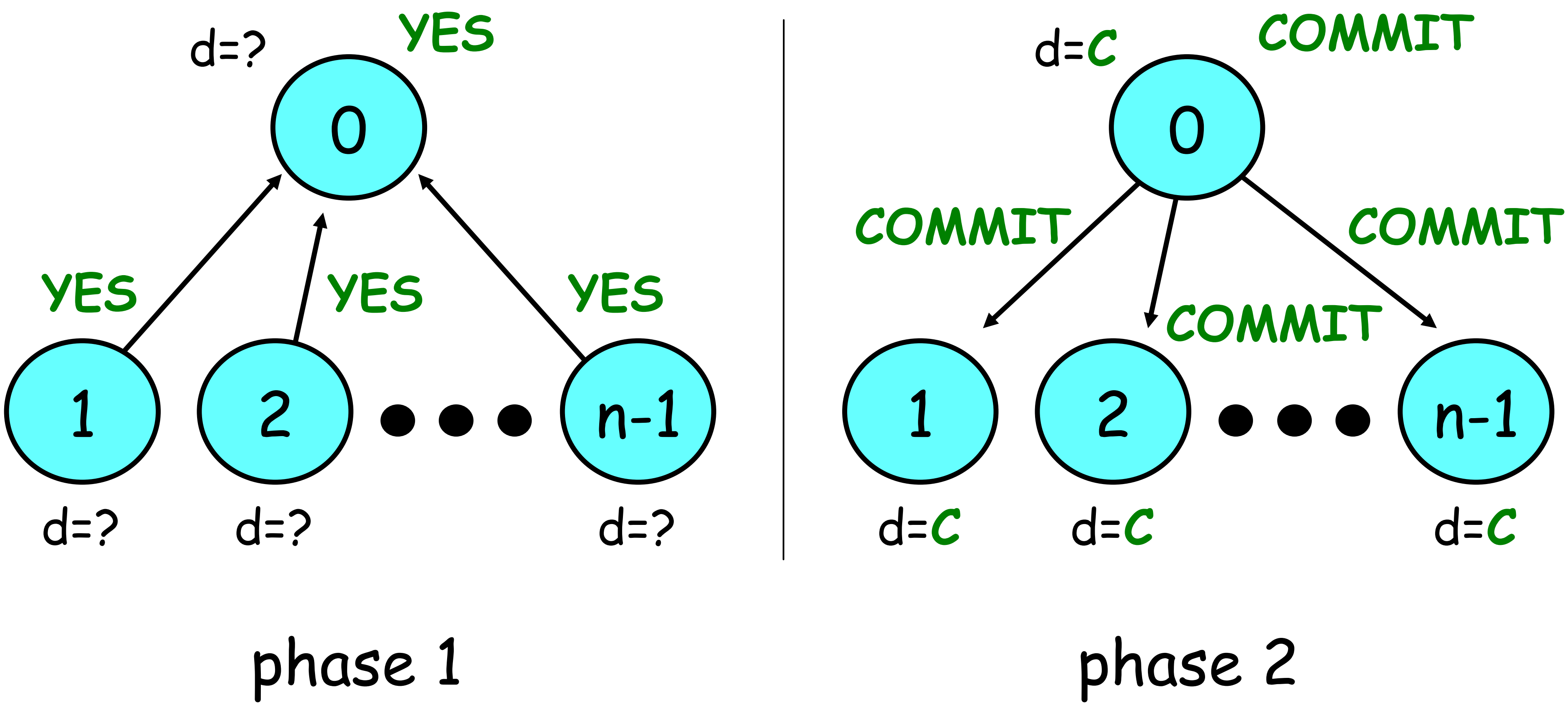
  abort: no participants make updates permanent

- Commit decision is based on unanimity of “yes” opinions among all participants

  if any participant has the opinion “no”, then commit is immediately not possible

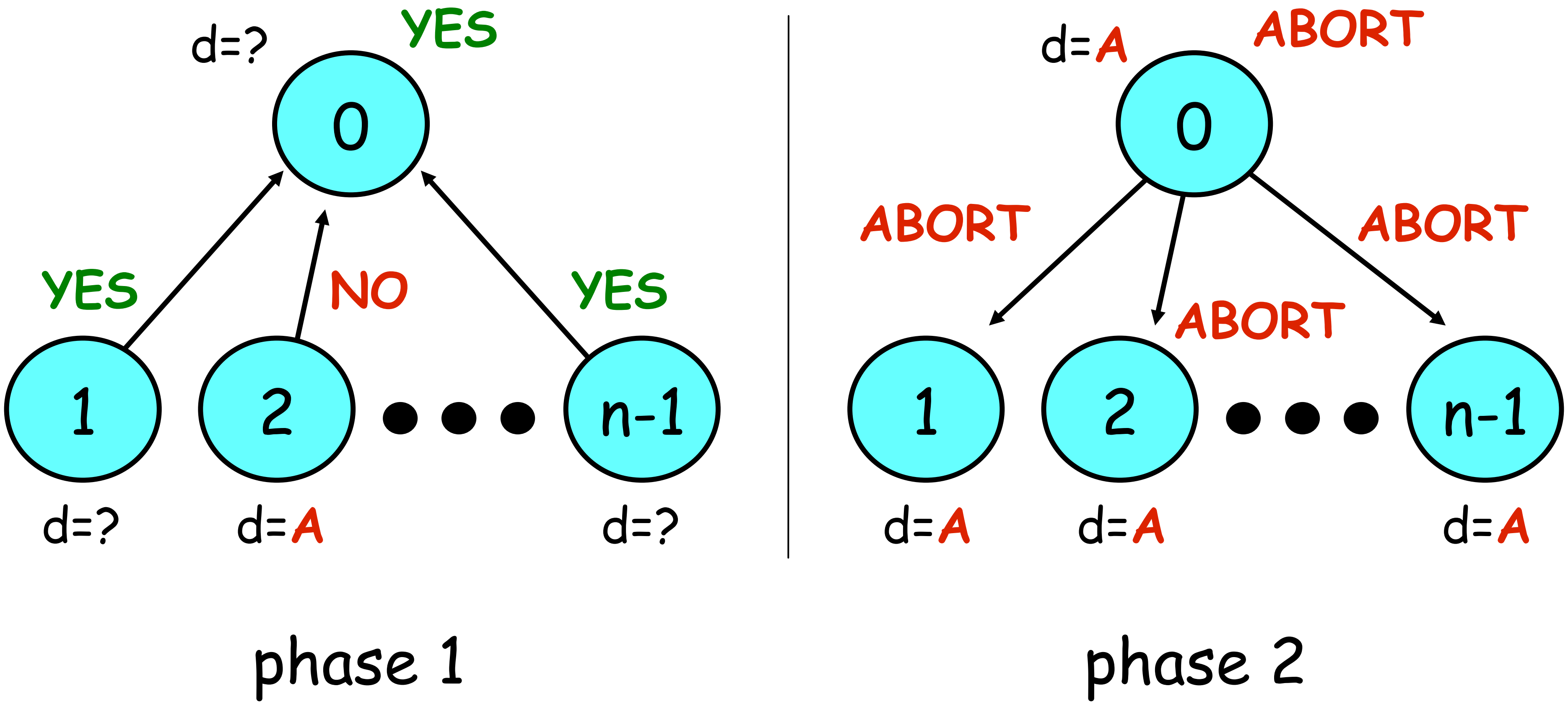
Familiar solution

2PC: two-phase atomic commitment protocol



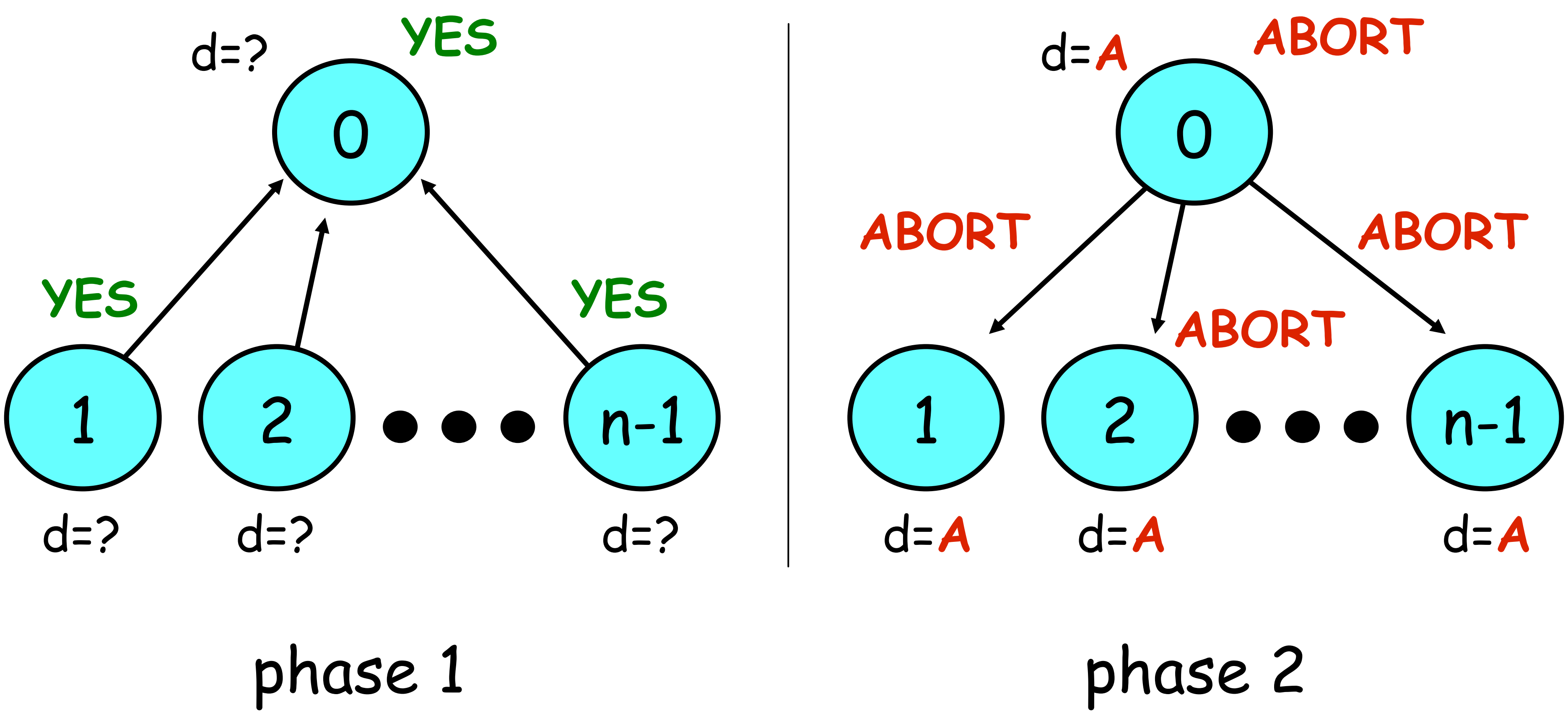
Familiar solution

2PC: two-phase atomic commitment protocol



Familiar solution

2PC: two-phase atomic commitment protocol



Two-phase commit

the desired outcome, formally

- An atomic commitment protocol satisfies the following properties

  AC1: all participants that decide reach the same decision

  AC2: if any participant decides "commit", then all participants must have voted "yes"

  AC3: if all participants vote "yes" and no failures occur, then all participants decide "commit"

  AC4: each participant decides at most once

Distributed transaction execution

```
  % some participant (the invoker) executes:
  send [T_START: trans, Δ_c, parts] to all parts

  % all participants (including invoker) execute:
  upon (receipt of [T_START: trans, Δ_c, parts])
    t_know := local_time
  % perform operations requested by transaction
    . . .
    if (willing/able to update permanently) then
      vote := YES
    else vote := NO
  % decide COMMIT or ABORT for transaction
    atomic_commitment(trans, parts)
```

Generic atomic commitment protocol

```
atomic_commitment(trans, parts)
cobegin
% the coordinator executes:
  send [VOTE_REQUEST] to all parts
  set-timeout-to local_time + 2∂
  wait-for (receipt of [VOTE: vote] from all parts)
    if (all votes = YES) then
      broadcast(COMMIT,parts)
    else broadcast(ABORT,parts)
  on-timeout
    broadcast(ABORT,parts)

//
```

Generic atomic commitment protocol
(continued)

```
% all participants (including coordinator) execute:
set-timeout-to (t_know + Δ_c + ∂)
wait-for (receipt of [VOTE_REQUEST] from coord)
  send [VOTE: vote] to coord
  if (vote = NO) then
    decide ABORT  % can make unilateral decision
  else
    set-timeout-to (t_know + Δ_c + 2∂ + Δ_b)
    wait-for (delivery of decision message)
      if (decision is ABORT) then
        decide ABORT
      else decide COMMIT
    on-timeout
      decide using termination_protocol()
on-timeout
  decide ABORT
coend
```

A simple broadcast

```
broadcast(m,G)

% broadcaster executes:
send [DLV: m] to all processes in G
deliver m

% processes other than broadcaster in G execute:
upon (receipt of [DLV: m])
  deliver m
```

Note: *deliver* is a primitive that means "make a message available to the process"

A simple broadcast, formally

- A simple broadcast protocol satisfies the following properties

  B1: if a correct process broadcasts a message *m*, then all correct processes in *G* eventually deliver *m*

  B2: for any message *m*, each process in *G* delivers *m* at most once, and only if some process broadcasts *m*

  B3: there exists a known constant Δ_b such that if the broadcast of *m* is initiated at time *t*, no process in *G* delivers *m* after time $t + \Delta_b$

GenericACP+SimpleBroadcast = 2PC

	Round 1		Round 2	
	Step 1 (exec. msg)	Step 2 (exec. trans)	Step 1 (exec. msg)	Step 2 (exec. trans)
Coord.	send vote request to all Part.$_i$ $i \neq 0$	receive *null*	send *null*	receive vote$_i$ from all $i \neq 0$ if $\forall$ i: vote$_i$ = YES then decision$_0$:= COMMIT else decision$_0$:= ABORT
Part.$_i$	send *null*	receive vote request if vote$_i$ = NO then decision$_i$:= ABORT	send vote$_i$	receive *null*

GenericACP+SimpleBroadcast = 2PC

Round 3

	Step 1 (exec. msg)	Step 2 (exec. trans)
Coord.	broadcast $decision_0$	receive *null*
$Part._i$	send *null*	receive $decision_0$ if $vote_i$ = YES then $decision_i$:= $decision_0$

Proof of correctness for ACP+SB

- ACP+SB is correct if it achieves properties AC1-AC4
- The proof method involves the use of the *coordinator axioms* AX1-AX3, the *broadcast properties* B1-B3, and the "*code*" of the protocol
- Proof proceeds in the order: AC2, AC3, AC4, AC1

ACP+SB achieves AC2

- If any participant decides "commit", then all participants must have voted "yes"
  - assume some participant *p* decides "commit"
  - this decision can only occur on line 30, so *p* must have delivered a commit message on line 27
  - by B2, "commit" was broadcast by some participant
  - this broadcast can only occur at line 16, so coordinator must have received votes from all participants, and those votes must all have been "yes"

ACP+SB achieves AC3

- If all participants vote "yes" and no failures occur, then all participants decide "commit"
  - assume all participants vote "yes" and no failures occur
  - let t_{start} be the time at which the transaction begins
  - by AX2 and AX3, coordinator C is chosen by $t_{start} + \Delta_c$
  - C sends vote requests, which arrive by $t_{start} + \Delta_c + \delta$

ACP+SB achieves AC3
(continued)

at line 21, participants wait for a vote request with timeout $\Delta_c + \delta$ time units after the time t_{know} at which they received the transaction at line 2

each participant receives the vote request before the timeout expires, since $t_{start} \leq t_{know}$

thus, all participants send their "yes" votes to C at line 22

votes arrive at C within 2δ time units of vote request, so timeout at line 13 does not expire

ACP+SB achieves AC3
(continued)

all votes are "yes", so C broadcasts commit message by time $t_{start} + \Delta_c + 2\delta$

all participants wait for decision message at line 27 with a timeout of $t_{know} + \Delta_c + 2\delta + \Delta_b$

by B1 and B3, every participant delivers the broadcast commit message by $t_{start} + \Delta_c + 2\delta + \Delta_b$ and so before the timeout expires ($t_{start} \leq t_{know}$)

therefore, all participants decide "commit" at line 30

ACP+SB achieves AC4

- Each participant decides at most once

  from the structure of the protocol, each participant decides at most once while executing (i.e., all paths from beginning to end *go* through a single “decide” action)

ACP+SB achieves AC1

- All participants that decide reach the same decision

  by contradiction

  suppose p decides commit and q decides abort

  by $AC4$, $p \neq q$

  q can only decide "abort" on line 24, 29, or 34

  by $AC2$, since p decides "commit", the coordinator C must have received "yes" votes from all, including q

ACP+SB achieves AC1
(continued)

since q sent a "yes" vote, it could not have decided "abort" on lines 24 or 34

so it must have decided "abort" on line 29, following delivery of an abort message on line 27

by B2, some participant C' must have broadcast the abort message, and that participant must have assumed the role of coordinator

by the protocol, a coordinator broadcasts only one decision, so $C \neq C'$, which contradicts AX1

Fundamental problem with ACP+SB

- What happens if, after a vote, there is no decision?

```
set-timeout-to (t_know + Δ_c + 2∂ + Δ_b)
wait-for (delivery of decision message)
  if (decision is ABORT) then
    decide ABORT
  else decide COMMIT
on-timeout
  decide using termination_protocol()
```

- Participant uses a *termination protocol*

  informally: contacts other participants in attempt to make a decision

What's wrong with termination?

- Correct participants may not be able to decide
- Consider the following scenario

  coordinator crashes during broadcast of decision

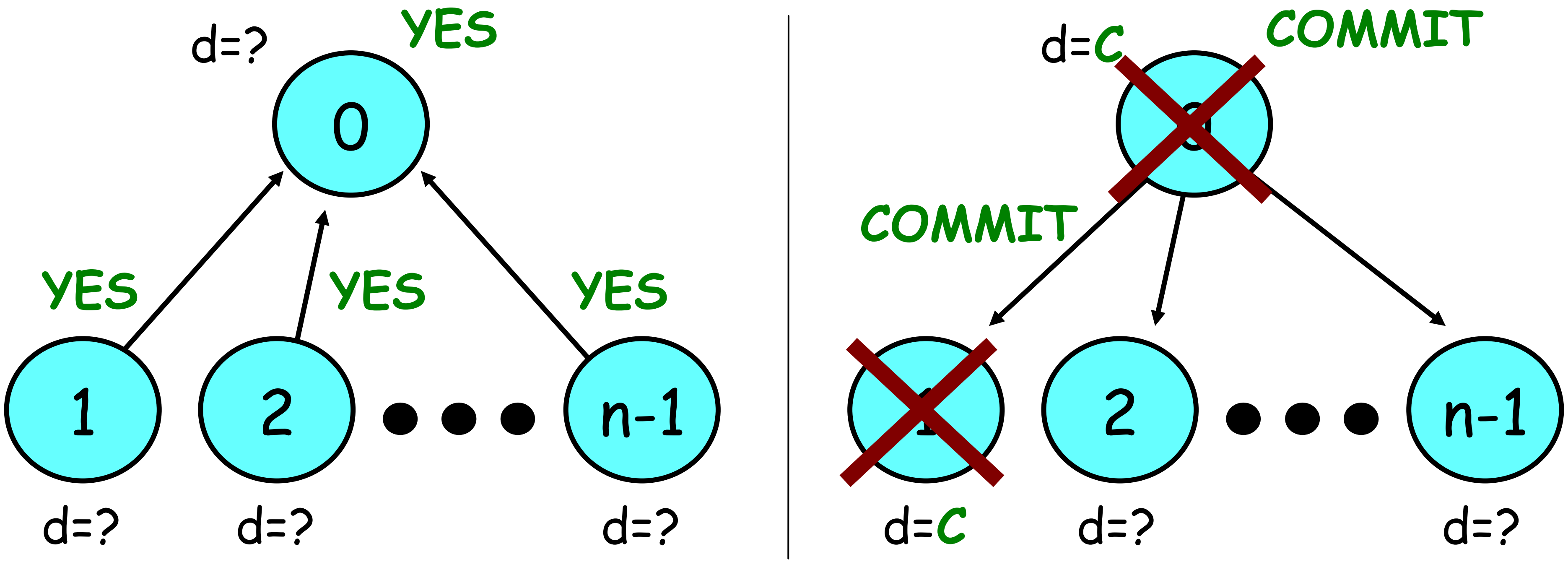
  (faulty) participants deliver the decision, then crash

  (correct) participants voted "yes", but do not deliver the decision (why "yes" and not "no"?)

  until *faulty* participants recover (if ever), any decision taken by *correct* participant may contradict decision taken by a *crashed* participant: *so wait*

Familiar solution

2PC: two-phase atomic commitment protocol



phase 1 phase 2

Blocking and non-blocking protocols

- An atomic commitment protocol is *blocking* if it admits executions in which correct participants cannot decide

    ACP+SB is a blocking protocol

    blocking results in poor use of resources

- An atomic commitment protocol is *non-blocking* if it satisfies AC1-AC4, as well as...

    AC5: every *correct* participant (one that never goes down during the transaction) eventually decides

How do we avoid blocking?

- Use a *stronger* broadcast protocol
- To the simple protocol we add another property

  B4: if any process (correct or not) in *G* delivers a message *m*, then all *correct* processes in *G* eventually deliver *m*

- Notice: this requires agreement among all processes, not just those that are correct

  a so-called *uniformity* property

UTRB: Uniform Timed Reliable Bcast

```
broadcast(m,G)

% broadcaster executes:
send [DLV: m] to all processes in G
deliver m

% processes other than broadcaster in G execute:
upon (receipt of [DLV: m])
upon (first receipt of [DLV: m])
  send [DLV: m] to all processes in G
  deliver m
```

- How does this satisfy B4?

  any process does not deliver *m* unless it previously relayed *m* to *all*, so *correct* processes will deliver *m*

ACP+UTRB: non-blocking commit

- Replace the broadcast protocol
- No need for termination protocol

  in a sense, built into the broadcast protocol

  safe to unilaterally decide “abort”

```
        set-timeout-to (t_know + Δ_c + 2∂ + Δ_b)
        wait-for (delivery of decision message)
          if (decision is ABORT) then
            decide ABORT
          else decide COMMIT
        on-timeout
          decide using termination_protocol()
          decide ABORT
```

ACP+UTRB achieves AC5

- Every correct participant eventually decides

  in ACP+SB, a correct participant could be prevented from reaching a decision only by executing the termination protocol on line 32

  all other decide actions have timeouts in *wait-for* statements that make indefinite waiting impossible

  ACP+UTRB makes a unilateral abort decision at line 32, eliminating the only source of blocking in the protocol

What if a process recovers?

- A participant that is *down* may become *operational* again after being *repaired*

  participant uses a *recovery protocol* to restore its local state and conclude any in-progress transactions

- Add the following property, if recovery possible

  AC6: if all participants that know about the transaction remain operational long enough, then they all decide

What is the nature of this recovery?

- Recovery protocol consists of two parts
- Actions performed by a recovering participant
  - could use a log file saved on stable storage
  - tries to unilaterally make a decision based on log
  - if unable, sends "help" messages to others
- Actions performed by the other participants
  - sends decision or "don't know"
- If decision received, then decide

Performance of ACP+UTRB

- Time and message complexity can be expressed as sum of ACP and UTRB costs
- Let n be the number of participants and let F be the maximum number of faulty participants

	ACP contribution	*UTRB contribution*
time complexity	2∂	Δ_b
message complexity	$2n$	μ_b

it can be shown that: $\Delta_b = (F + 1)\partial$ *and* $\mu_b = n^2$

Message-Optimized UTRB

- Message complexity can be reduced from quadratic to linear

- Idea: *rotating broadcasters*

  rather than every process relaying every message to all other processes under all circumstances, "backup" processes (called *cohorts*) assume the role of broadcaster, in turn, as failures occur

Message-Optimized UTRB

basic structure

```
    broadcast(m,G)

  % broadcaster executes:
    send [MSG: m,cohorts,1] to all processes in G
    send [DLV: m] to all processes in G

  % processes p in G execute:
    upon (first receipt of [MSG: m,cohorts,index])
    ...
         set-timeout-to ...
         wait-for (receipt of [DLV: m])
    ...
         on-timeout
    ...
               send [REQ: m,cohorts,i] to chorts[i]
    ...
    upon (first receipt of [REQ: m,cohorts,index])
    ...
```

Message-Optimized UTRB

in detail

```
upon (first receipt of [MSG: m,cohorts,index])
  i := index
  first_timeout := local_clock + ∂
  for k := 0,1, ... do
     set-timeout-to first_timeout + k∂
     wait-for (receipt of [DLV: m])
        deliver m
        exit loop
     on-timeout
        if (i < F + 1) then
           i := i + 1
           send [REQ: m,cohorts,i] to chorts[i]
        else exit loop
  end loop
upon (first receipt of [REQ: m,cohorts,index])
  send [MSG: m,cohorts,index] to all processes in G
  send [DLV: m] to all processes in G
```

Performance of MOUTRB

- Can consider the *actual* number of faulty processes f (of course, f will be ≤ F, the max)
- Each cohort adds 2δ time units and $2n$ messages
- Linear increase in both dimensions

Num faulty processes	Δ_b	μ_b
$f = 0$	2∂	$2n$
$1 \leq f \leq F$	$(f + 1)2\partial$	$(f + 1)2n$

How might time complexity improve?

- Observe: phases of REQ and DLV messages as cohorts fail in turn
- Idea: send REQ messages while still waiting for previous-phase DLV messages to arrive

    anticipate failure of previous cohort

    reduce timeout from 2δ to δ
- With no failures, complexity same as MOUTRB
- With failures, increase in message traffic

Decentralized 2PC

- Recall that the original model assumes a "centralized" coordination step to decide the outcome of the transaction

- What if we *decentralized* (i.e., "distributed") the role of the coordinator?

Decentralized 2PC

Round 1

	Step 1 (exec. msg)	Step 2 (exec. trans)
$Part._i$	send $vote_i$ to all $Part._j$ $j \neq i$	receive $vote_j$ from all $j \neq i$ if $vote_i$ = NO or any vote received = NO then $decision_i$:= ABORT else if $vote_i$ = YES and $\forall$ $j \neq i$: $vote_j$ = YES then $decision_i$:= COMMIT

- Is D2PC a *blocking* or *non-blocking* protocol?
- What is the *complexity* of D2PC?