Prof Philippas Tsigas
Distributed Computing and Systems Research Group

DISTRIBUTED SYSTEMS II

FAULT-TOLERANT AGREEMENT II

Conditions for a solution for Byzantine faults

- Number of processes: **n**
- Maximum number of possibly failing processes: **f**
- **Necessary and sufficient condition** for a solution to Byzantine agreement:

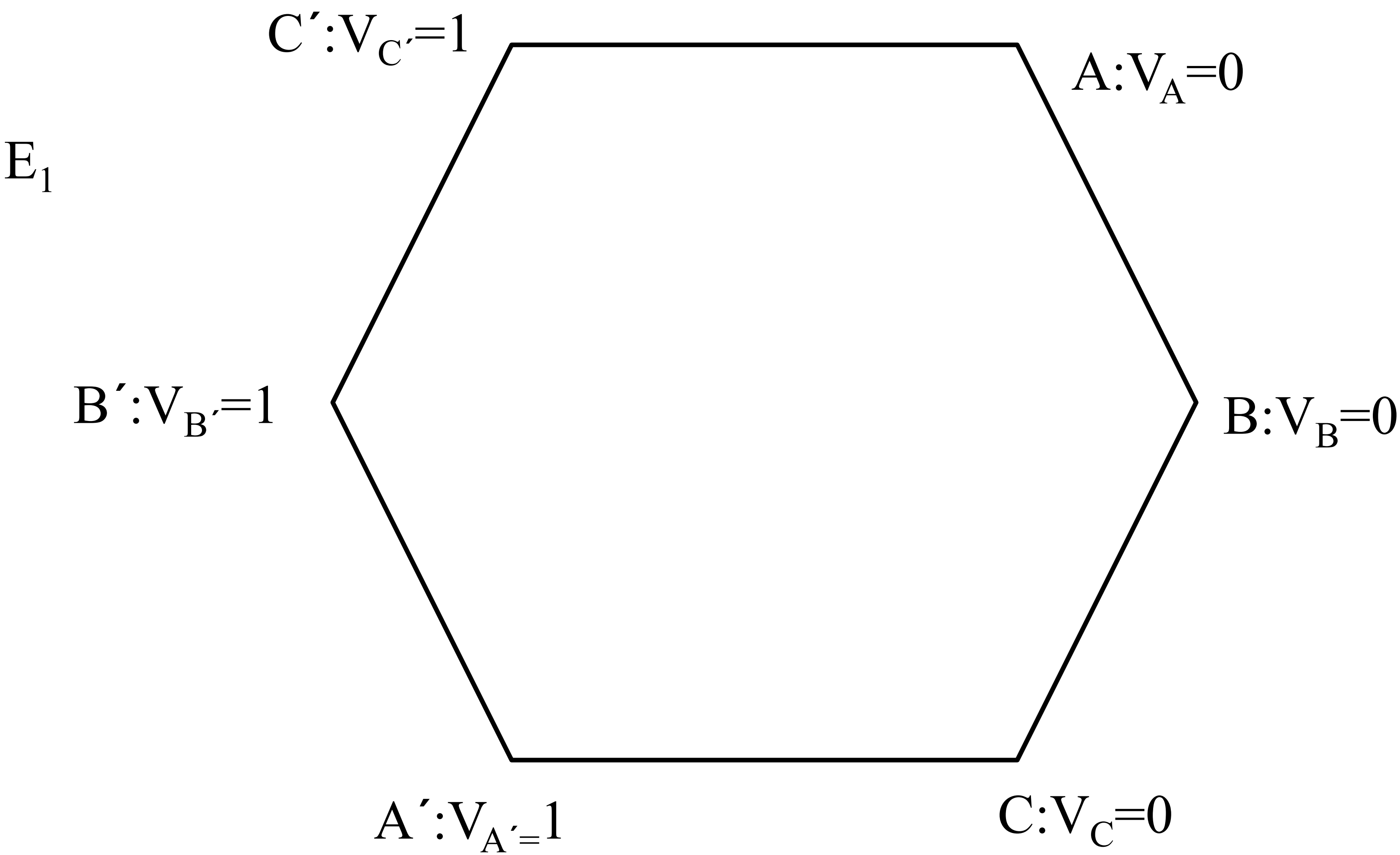
  $$f<n/3$$

- •**Minimal number of rounds** in a deterministic solution**:**

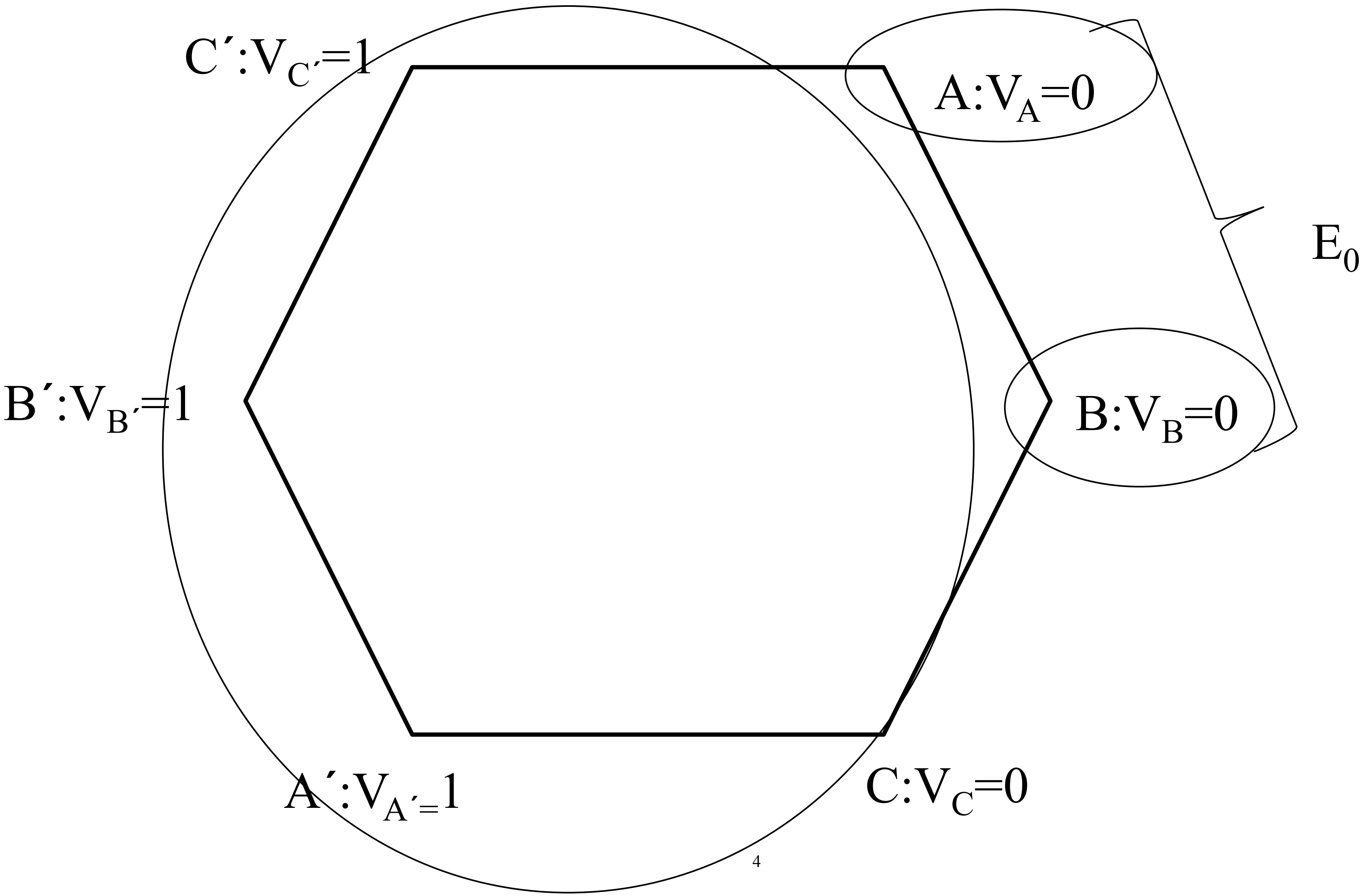
  $$f+1$$

- •There exist **randomized solutions with a lower expected number of rounds**

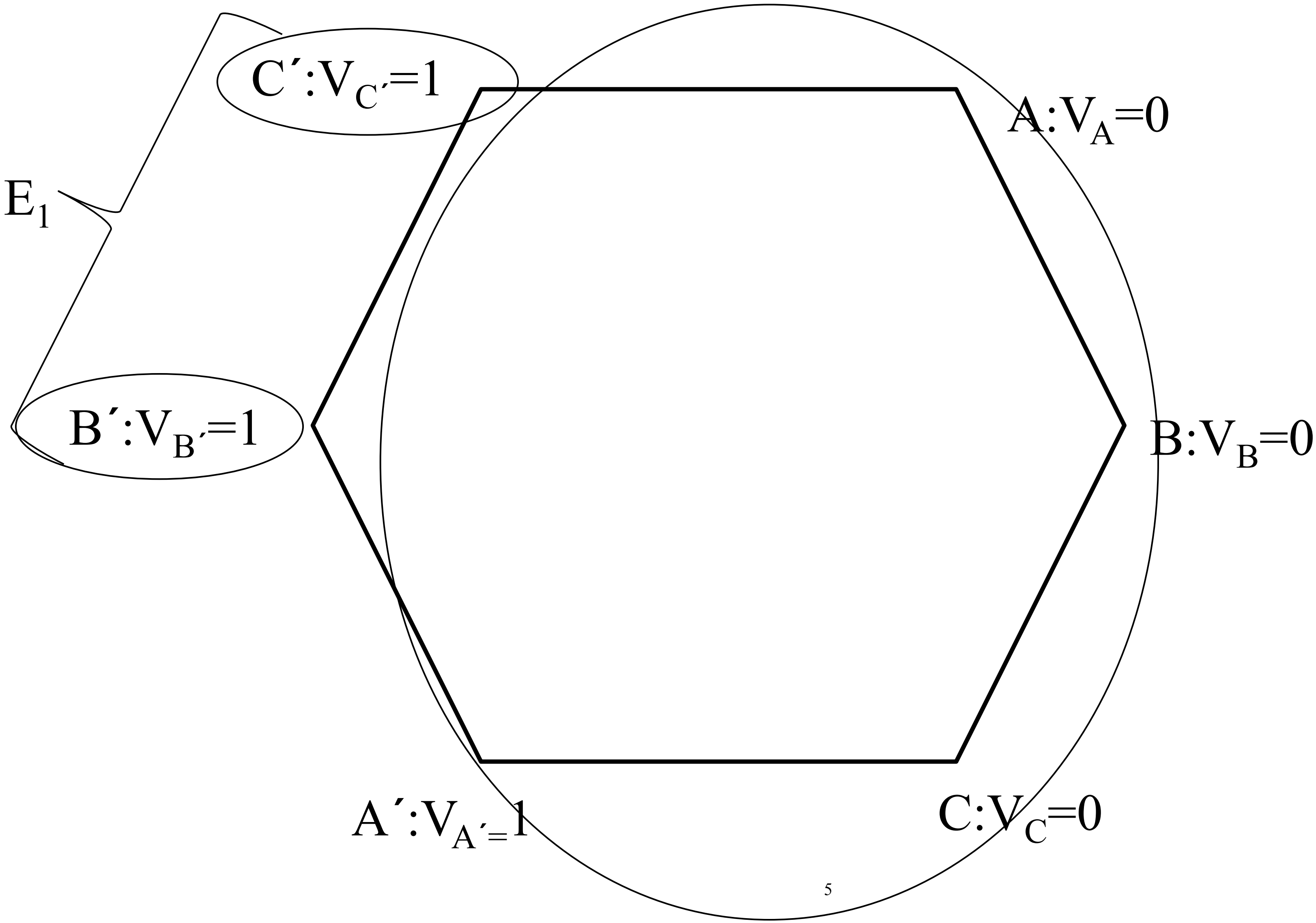
Impossibility of 1-resilient 3-processor Agreement



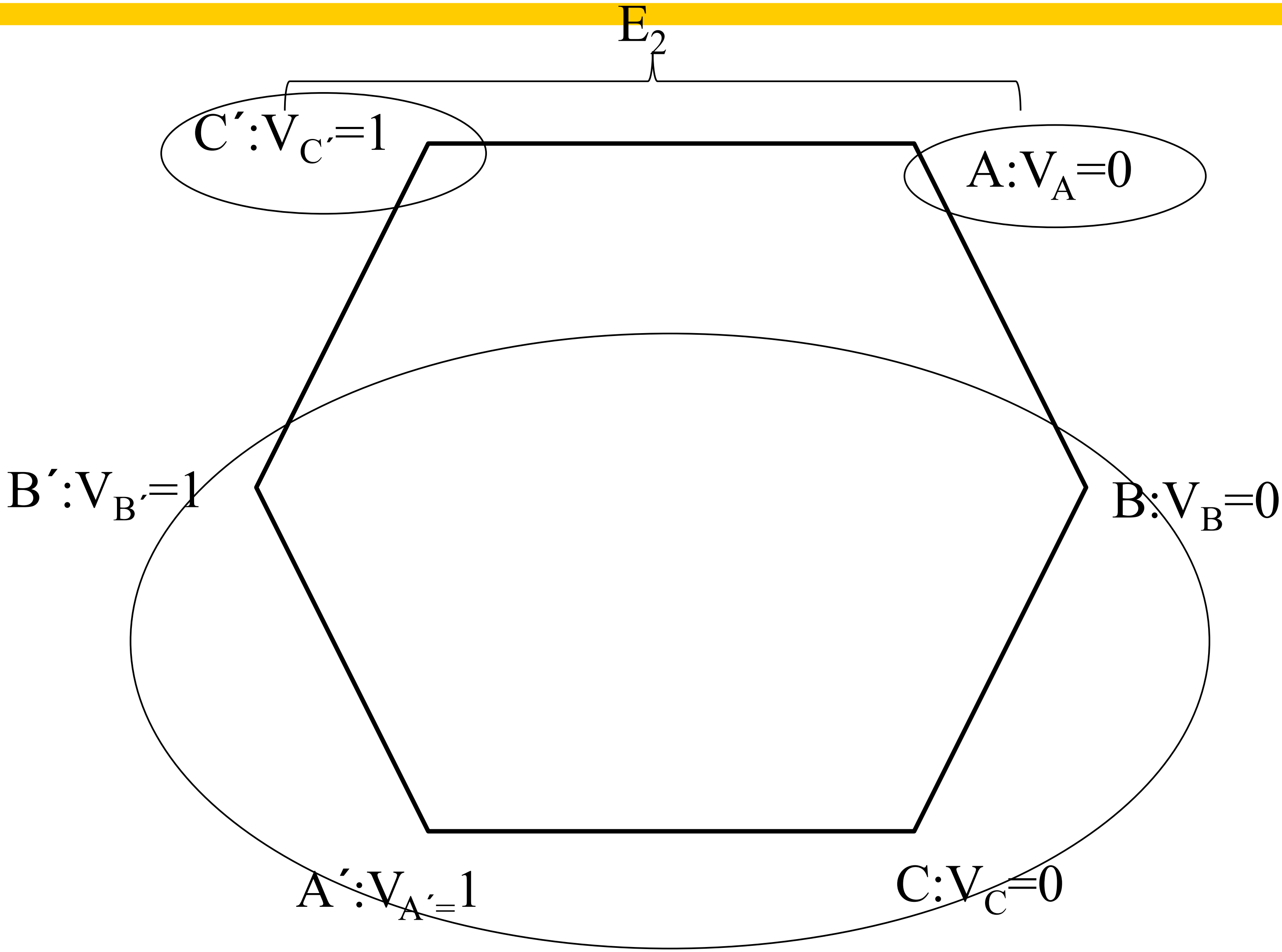
Impossibility of 1-resilient 3-processor Agreement



Impossibility of 1-resilient 3-processor Agreement



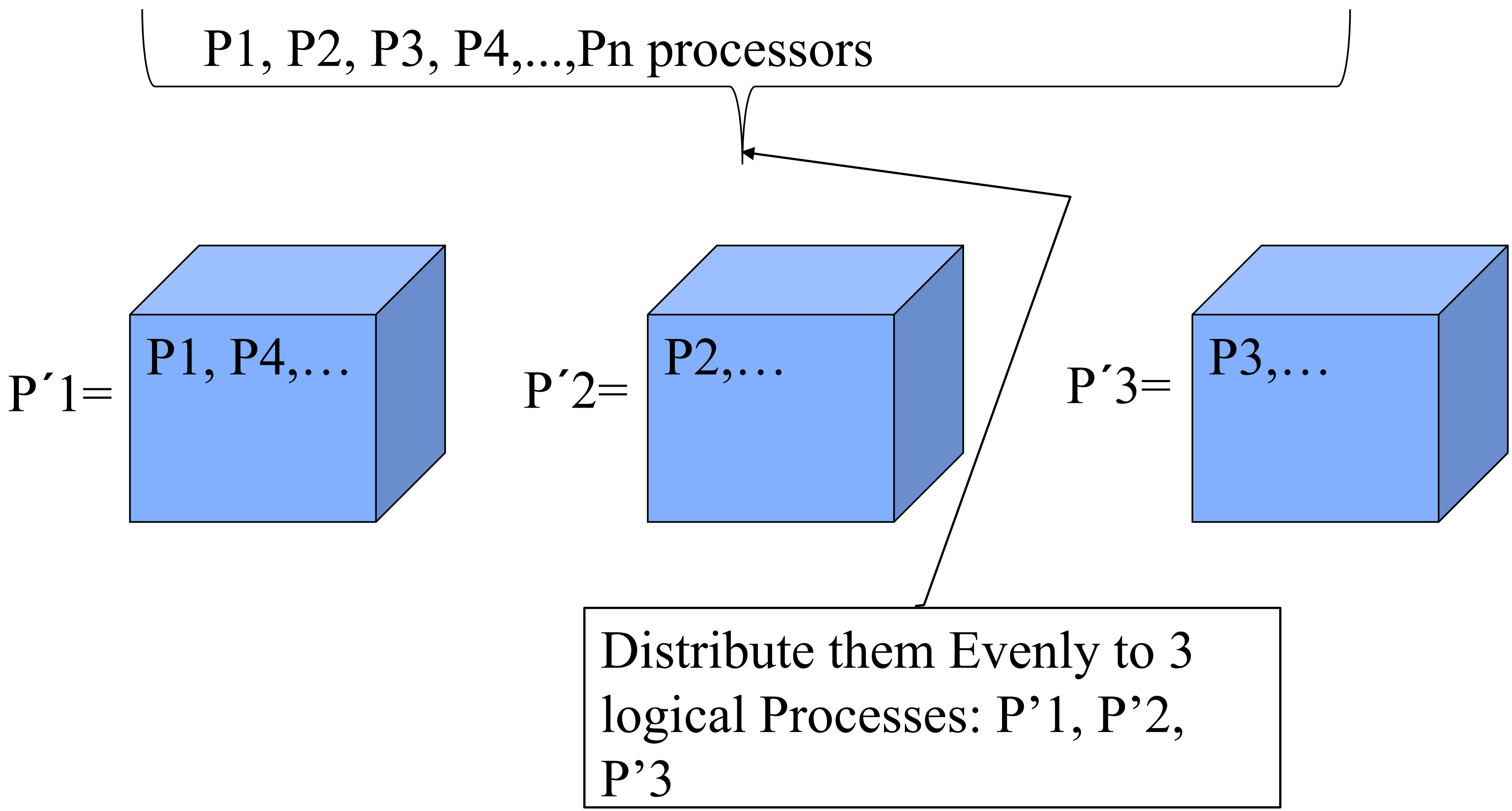
Impossibility of 1-resilient 3-processor Agreement



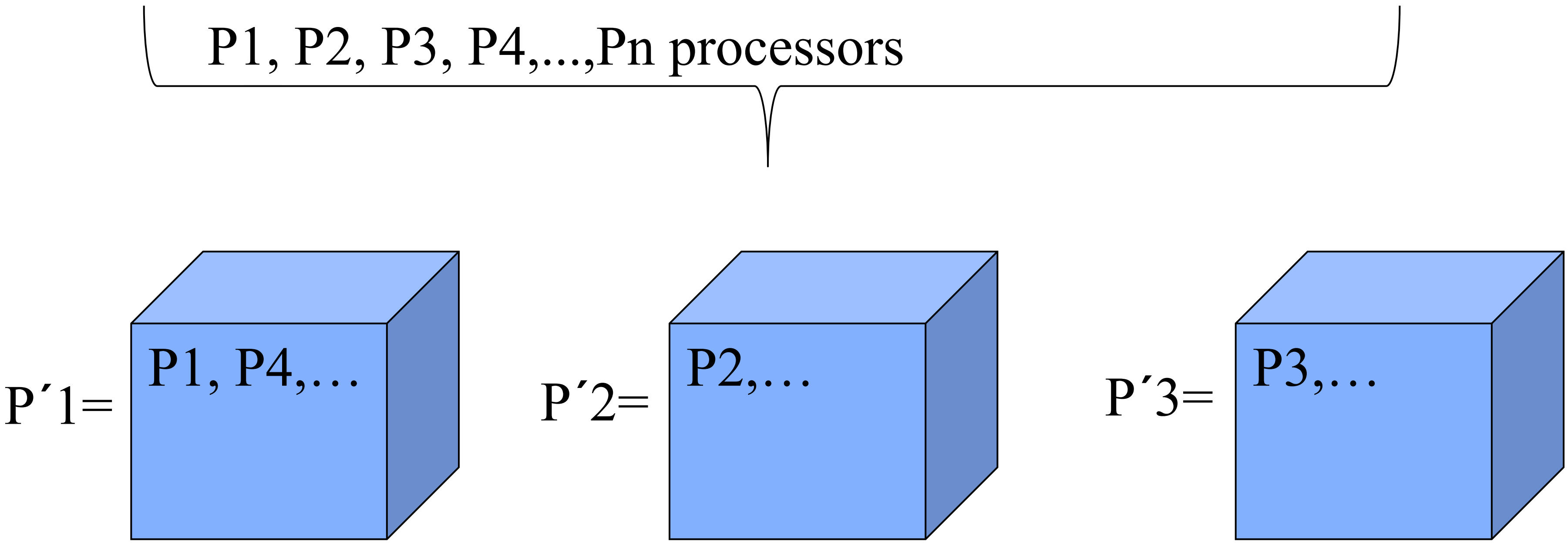
Proof

- In E_0 A and B decide 0
- In E_1 B´ and C´ decide 1
- In E_2 C´ has to decide 1 and A has to decide 0, contradiction!

t-resilient algorithm requiring n<=3t processors, t=>2

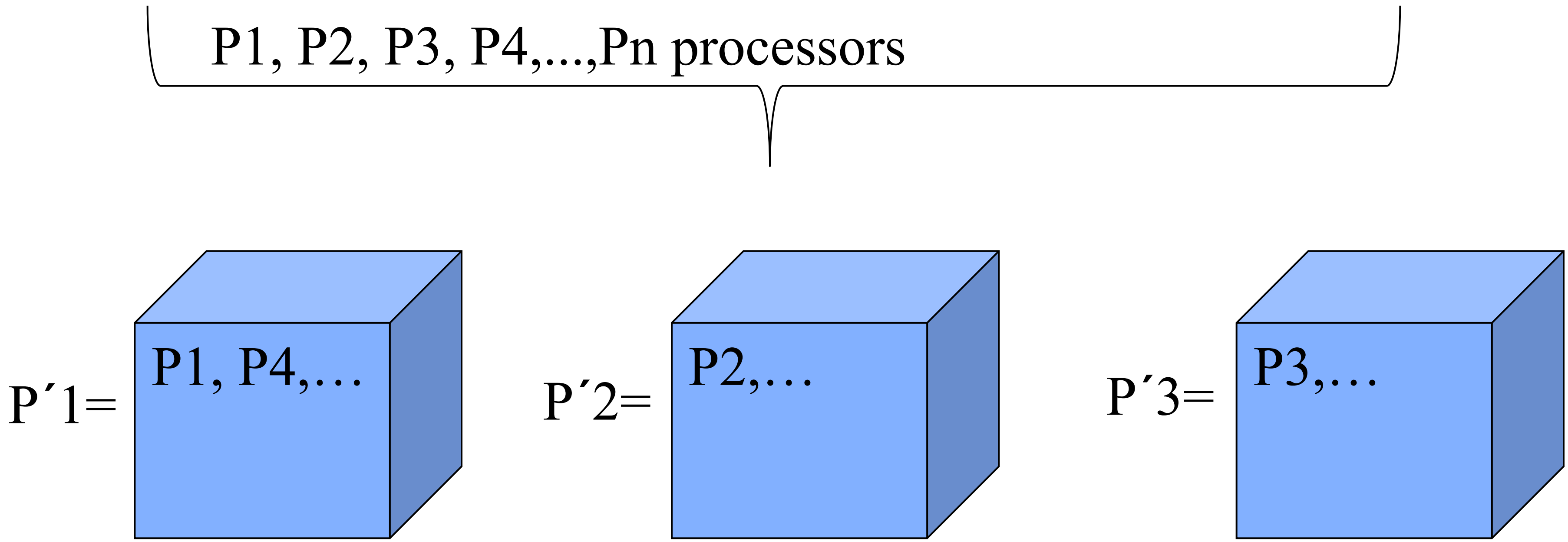


t-resilient algorithm requiring n<=3t processors, t=>2



|P´1|, |P´2|, |P´3| < t

t-resilient algorithm requiring n<=3t processors, t=>2



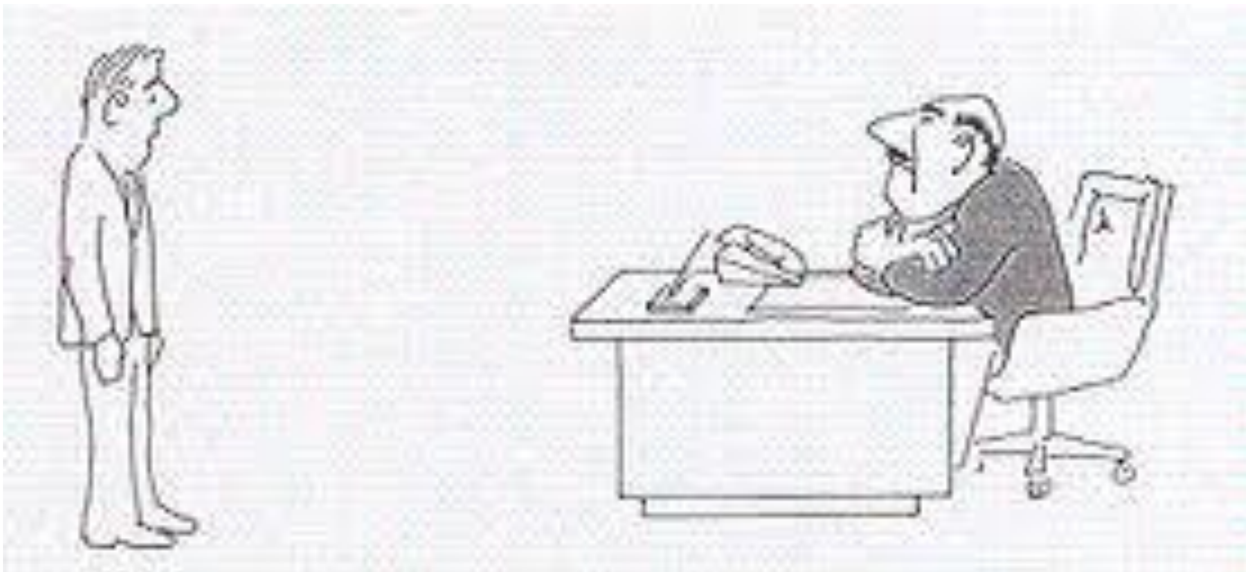
In the system with 3 processors (P´1, P´2 and P´3) if one of them is faulty then at most t processors of the initial system are going to be faulty.

Proof

- Run the solution to the problem (sysyem with n processes) at the 3 process system.
- Ask P'1, P'2 and P'3 to simulate their respective substem and decide what the processes of its subsystem have decided.
- Contradiction! This is a solution to a 3 processes system with one byzantine process.

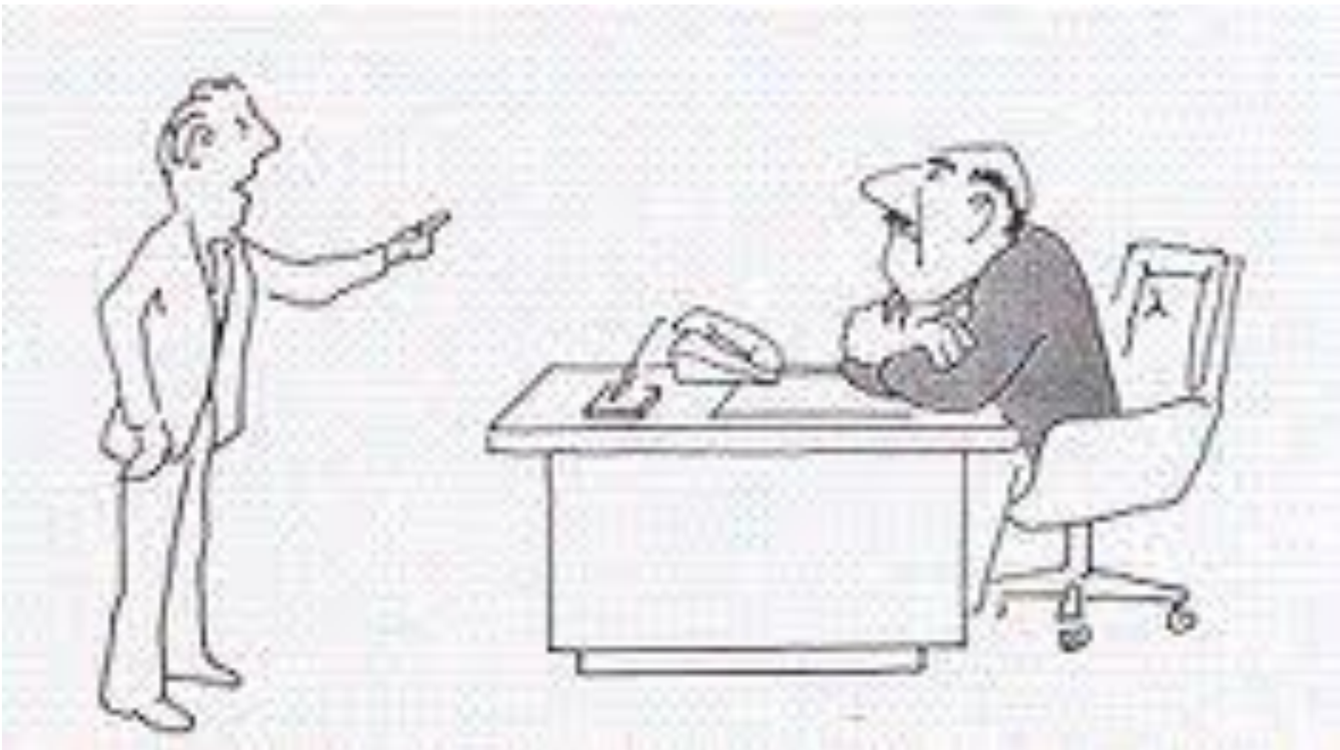
”I can’t find a solution, I guess I’m just too dumb”

- Picture from Computers and Intractability, by Garey and Johnson



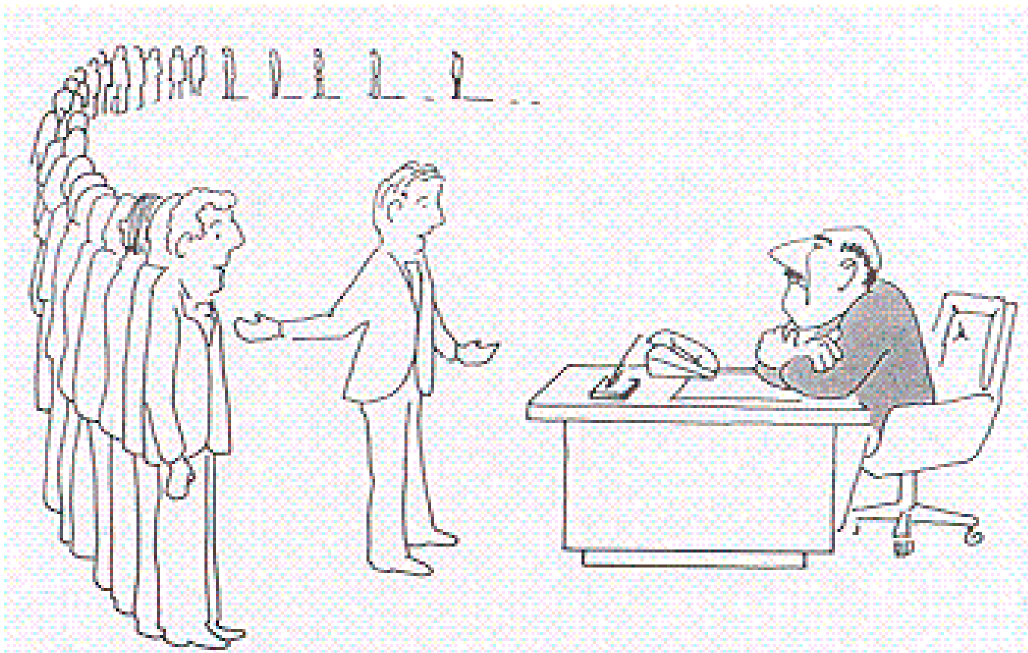
”I can’t find an algorithm, because no such algorithm is possible”

- Picture from Computers and Intractability, by Garey and Johnson



”I can’t find an algorithm, but neither can all these famous people.”

- Picture from Computers and Intractability, by Garey and Johnson



Consensus in a Synchronous System with process crashing

- For a system with at most *f* processes crashing, the algorithm proceeds in *f+1* rounds (with timeout), using basic multicast.

- $Values^r_i$: the set of proposed values known to P_i at the beginning of round *r*.
- Initially $Values^0_i = \{\}$; $Values^1_i = \{v_i\}$

for round = 1 to f+1 do

  multicast ($Values^r_i - Values^{r-1}_i$)

  $Values^{r+1}_i \leftarrow Values^r_i$

  for each V_j received

  $Values^{r+1}_i = Values^{r+1}_i \cup V_j$

  end

end

d_i = minimum($Values^{f+2}_i$)

Proof of Correctness

Proof by contradiction.

- Assume that two processes differ in their final set of values.
- Assume that p_i possesses a value *v* that p_j does not possess.
  - → A third process, p_k, sent *v* to p_i, and crashed before sending *v* to p_j.
  - → Any process sending *v* in the previous round must have crashed; otherwise, both p_k and p_j should have received *v*.
  - → Proceeding in this way, we infer at least one crash in each of the preceding rounds.
  - → But we have assumed at most *f* crashes can occur and there are $f+1$ rounds → contradiction.

Byzantine agreem. with authentication and Synchrony

- Every message **carries a signature**
- The signature of a loyal general **cannot be forged**
- Alteration of the contents of a signed message can be detected
- Every (loyal) general can **verify the signature of any other (loyal) general**
- **Any number f of traitors can be allowed**
- Commander is process **0**
- Structure of message from (and signed by) the commander, and subsequently signed and sent by lieutenants **Li1, Li2,…:**
- **(v : s0 : si1: … : sik)**
- Every lieutenant maintains **a set of orders V**
- Some choice function on **V for deciding (e.g., majority, minimum)**

- •Algorithm in commander:

  send**(v: s0)to every lieutenant**

  – Algorithm in every **lieutenant Li**:

  upon receipt of (v : s0: si1: …. : sik) do

  if (v not in V) then

  V := V union {v}

  if (k < f) then

  for(j in {1,2,…,n-1} \{i,i1,…,ik}) do

  send(v: s0: si1: … : sik: i) to Lj

  If (Li will not receive any more messages) then decide(choice(V))

Atomic commit protocols

- transaction atomicity requires that at the end,
  - either all of its operations are carried out or none of them.
- in a distributed transaction, the client has requested the operations at more than one server
- one-phase atomic commit protocol
  - the coordinator tells the participants whether to commit or abort
  - what is the problem with that?
  - this does not allow one of the servers to decide to abort – it may have discovered a deadlock or it may have crashed and been restarted
- two-phase atomic commit protocol
  - is designed to allow any participant to choose to abort a transaction
  - *phase 1* - each participant votes. If it votes to commit, it is *prepared.* It cannot change its mind. In case it crashes, it must save updates in permanent store
  - phase 2 - the participants carry out the joint decision

The decision could be *commit* or *abort* - participants record it in permanent store

Failure model for the commit protocols

- Failure model for transactions
  - this applies to the two-phase commit protocol
- Commit protocols are designed to work in
  - synchronous system, system failure when a msg does not arrive on time.
  - servers may crash but *a new process whose state is set from information saved in permanent storage and information held by other processes*.
  - messages may NOT be lost.
  - assume corrupt and duplicated messages are removed.
  - no byzantine faults – servers either crash or they obey their requests
- 2PC is an example of a protocol for reaching a consensus.
  - Chapter 11 says consensus cannot be reached in an asynchronous system if processes sometimes fail.
  - however, 2PC does reach consensus under those conditions.
  - because crash failures of processes are masked by replacing a crashed process with a new process whose state is set from information saved in permanent storage and information held by other processes.

Operations for two-phase commit protocol

canCommit?(trans)-> Yes / No

This is a request with a reply

Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.

doCommit(trans)

Call from coordinator to participant to tell participant to commit its part of a transaction.

These are asynchronous requests to avoid delays

doAbort(trans)

Call from coordinator to participant to tell participant to abort its part of a transaction.

Asynchronous request

haveCommitted(trans, participant)

Call from participant to coordinator to confirm that it has committed the transaction.

getDecision(trans) -> Yes / No

Call from participant to coordinator to ask for the decision on a transaction after it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages.

Figure 13.4

- participant interface- *canCommit?, doCommit, doAbort*

  coordinator interface- *haveCommitted, getDecision*

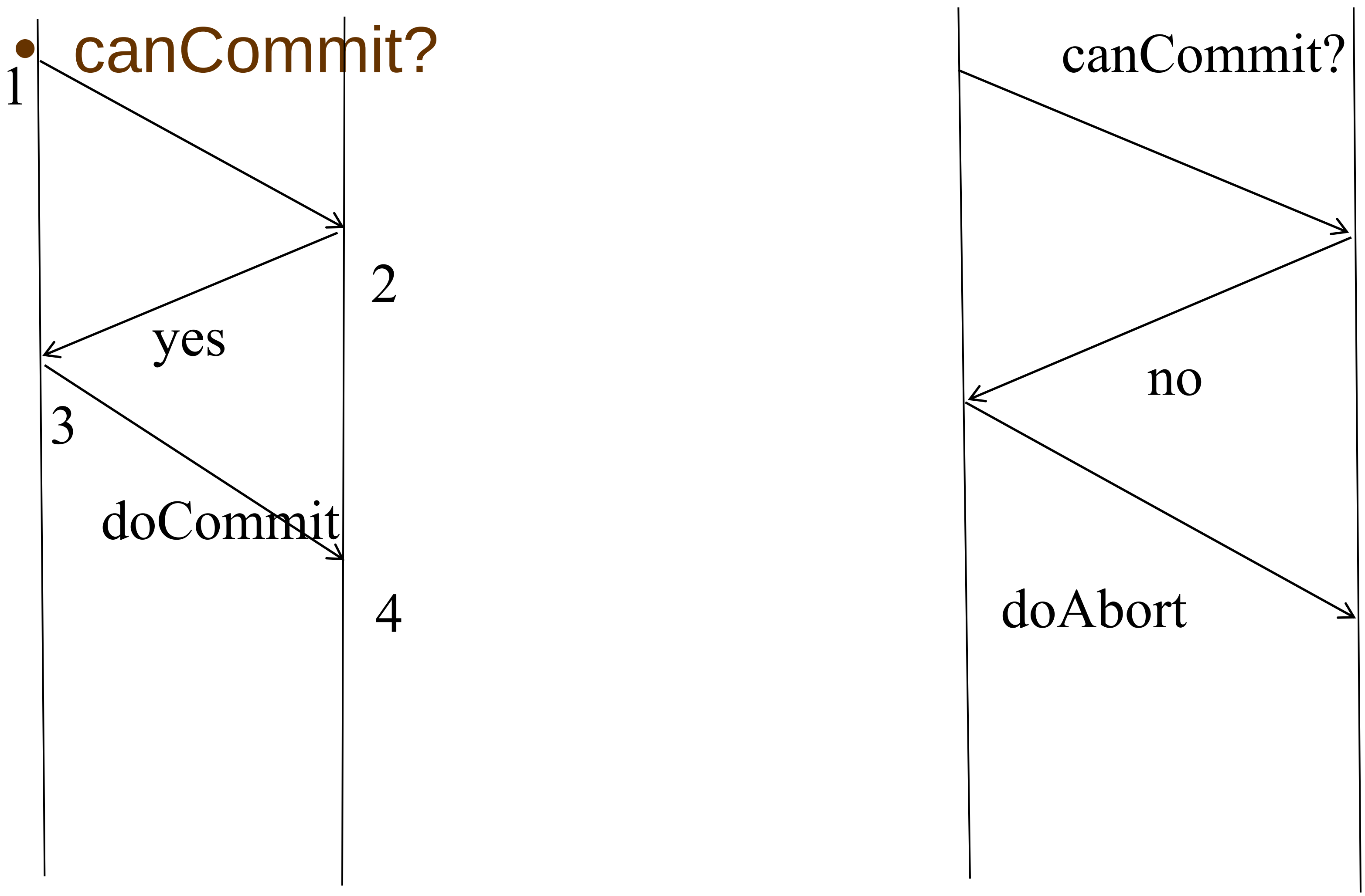
The two-phase commit protocol

- *Phase 1 (voting phase):*
  - 1. The coordinator sends a *canCommit*? request to each of the participants in the transaction.
  - 2. When a participant receives a *canCommit*? request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No* the participant aborts immediately.
- *Phase 2 (completion according to outcome of vote):*
  - 3. The coordinator collects the votes (including its own).
    - (a)If there are no failures and all the votes are *Yes* the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
    - (b)Otherwise the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.
  - 4. Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

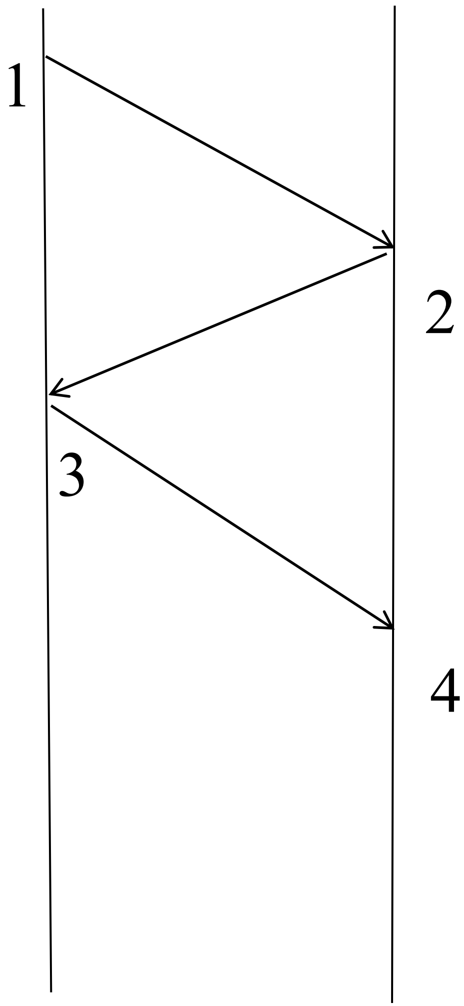
Figure 13.5

Two-Phase Commit Protocol

- canCommit?



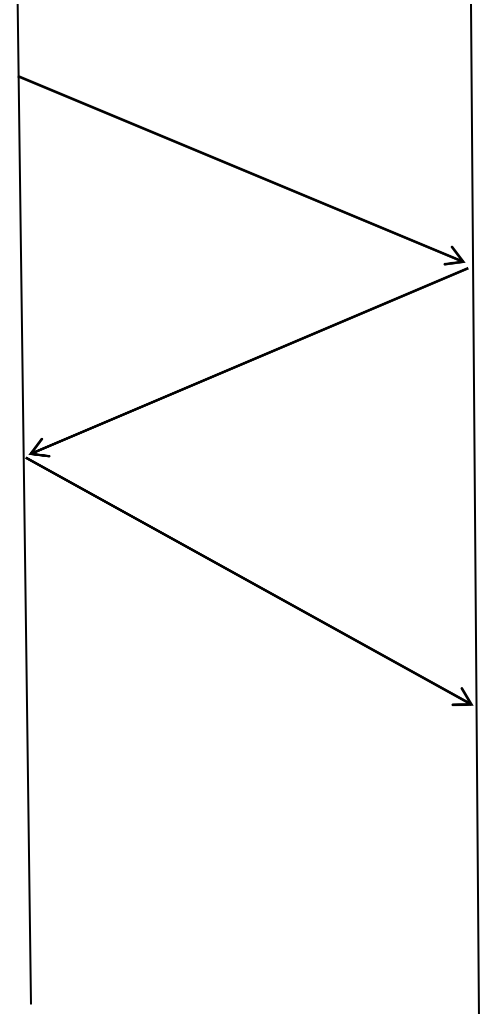
TimeOut Protocol



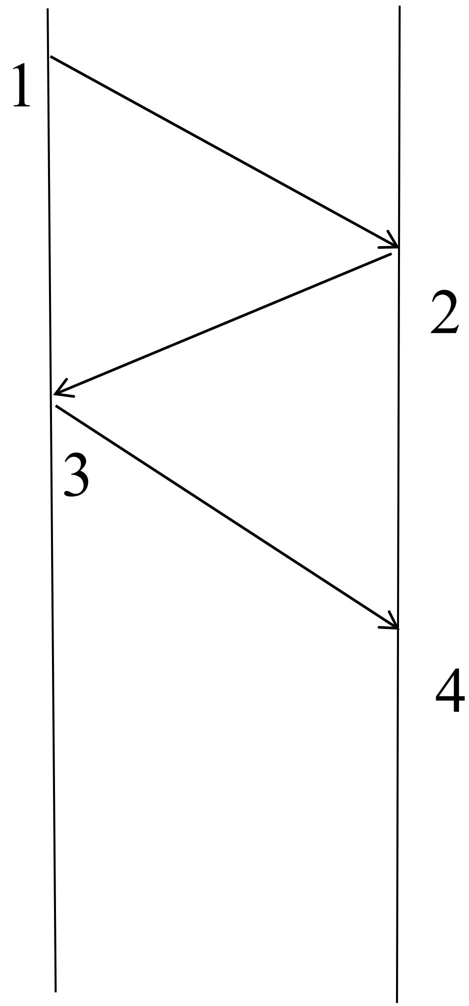
At step 2 and 3 no commit decision made

OK to abort

Coordinator will either not collect all commit votes or will vote for *abort*

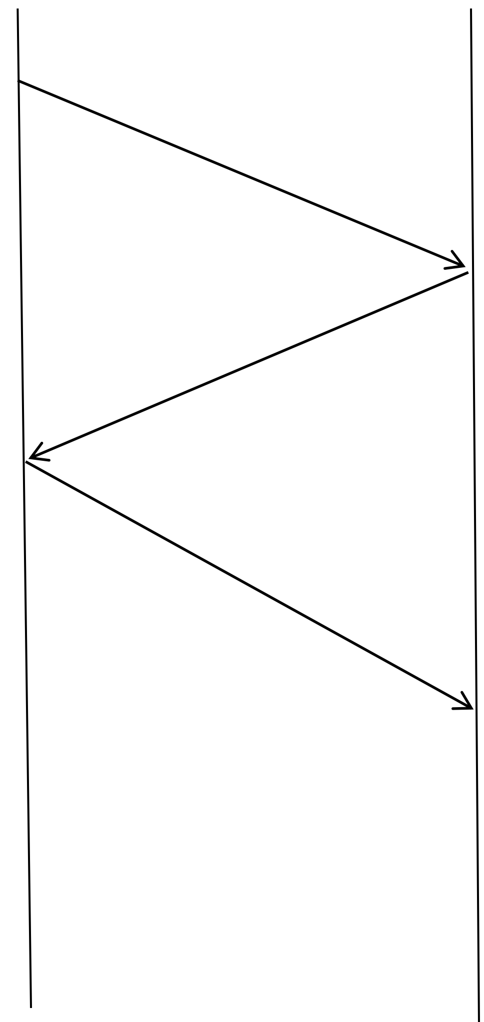


TimeOut Protocol

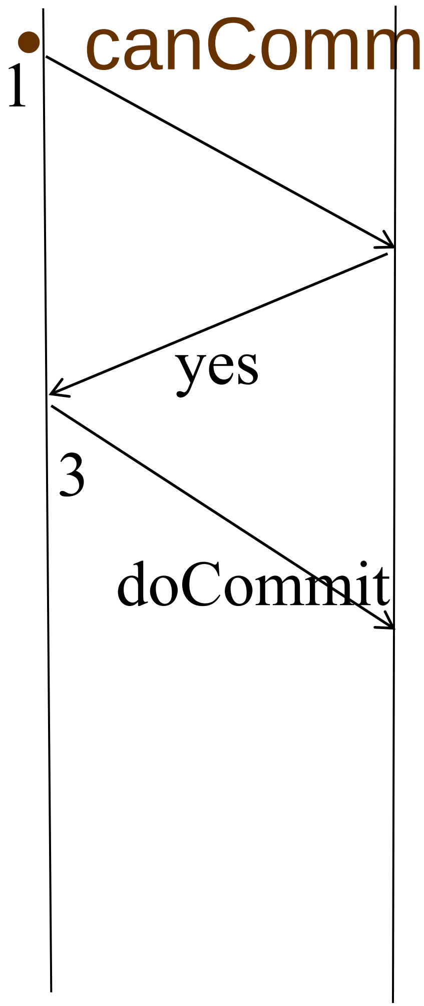


At step 4

- *cohort cannot communicate with coordinator*
- *Coordinator mayhave decided*
- *Cohort must block until communication re-established*
- *Might ask other cohorts*



Restart Protocol



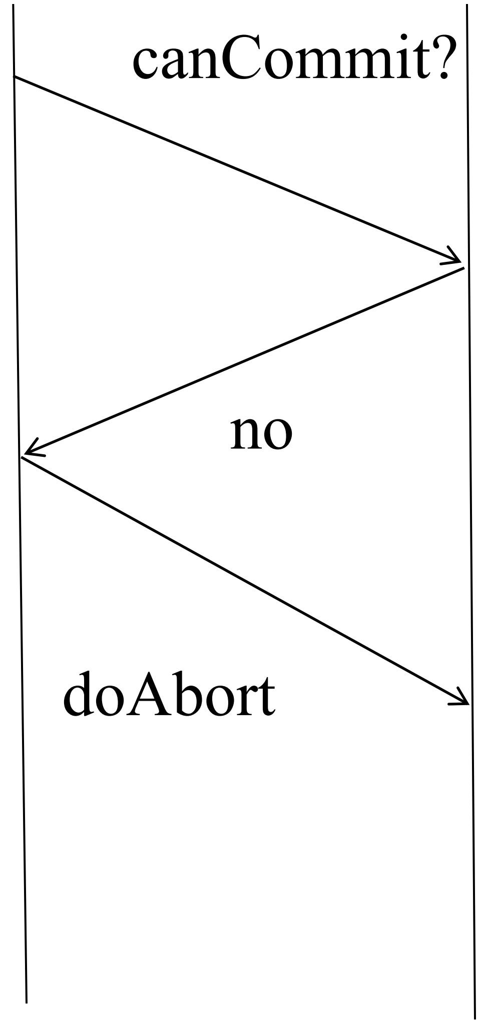
2

If the site

- Has decided, it just picks up from where it left off
- Is a cohort that had not voted, it decides abort
- Is a cordinator that has not decided, it decides abort

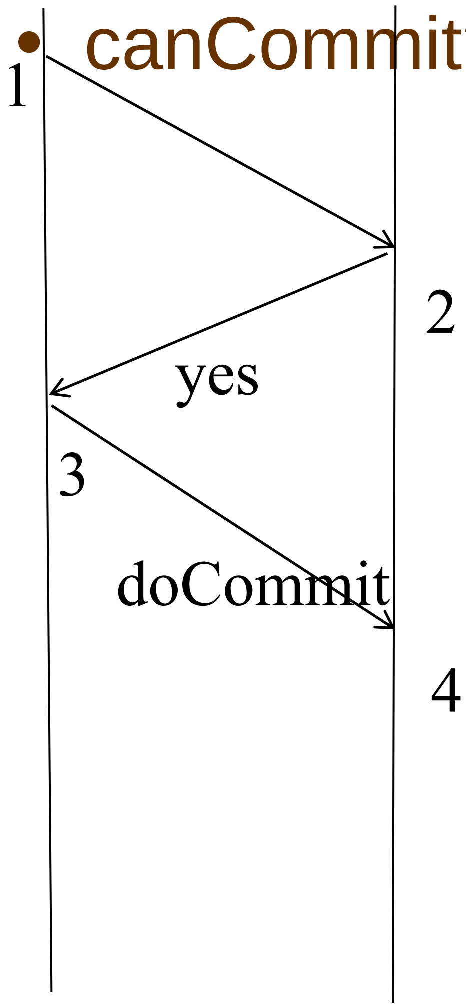
4

- A cohort that crashed after voting commit, it must block until it discovers



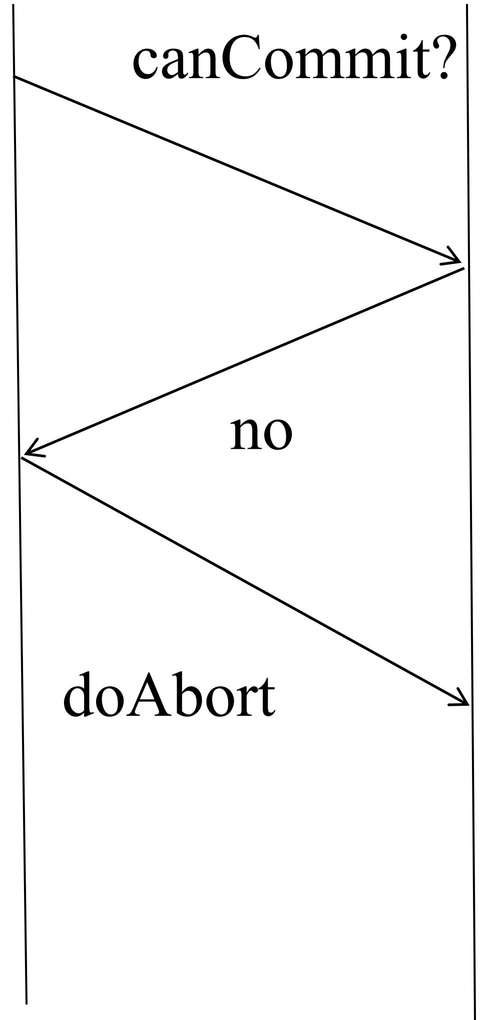
Blocking

- canCommit?



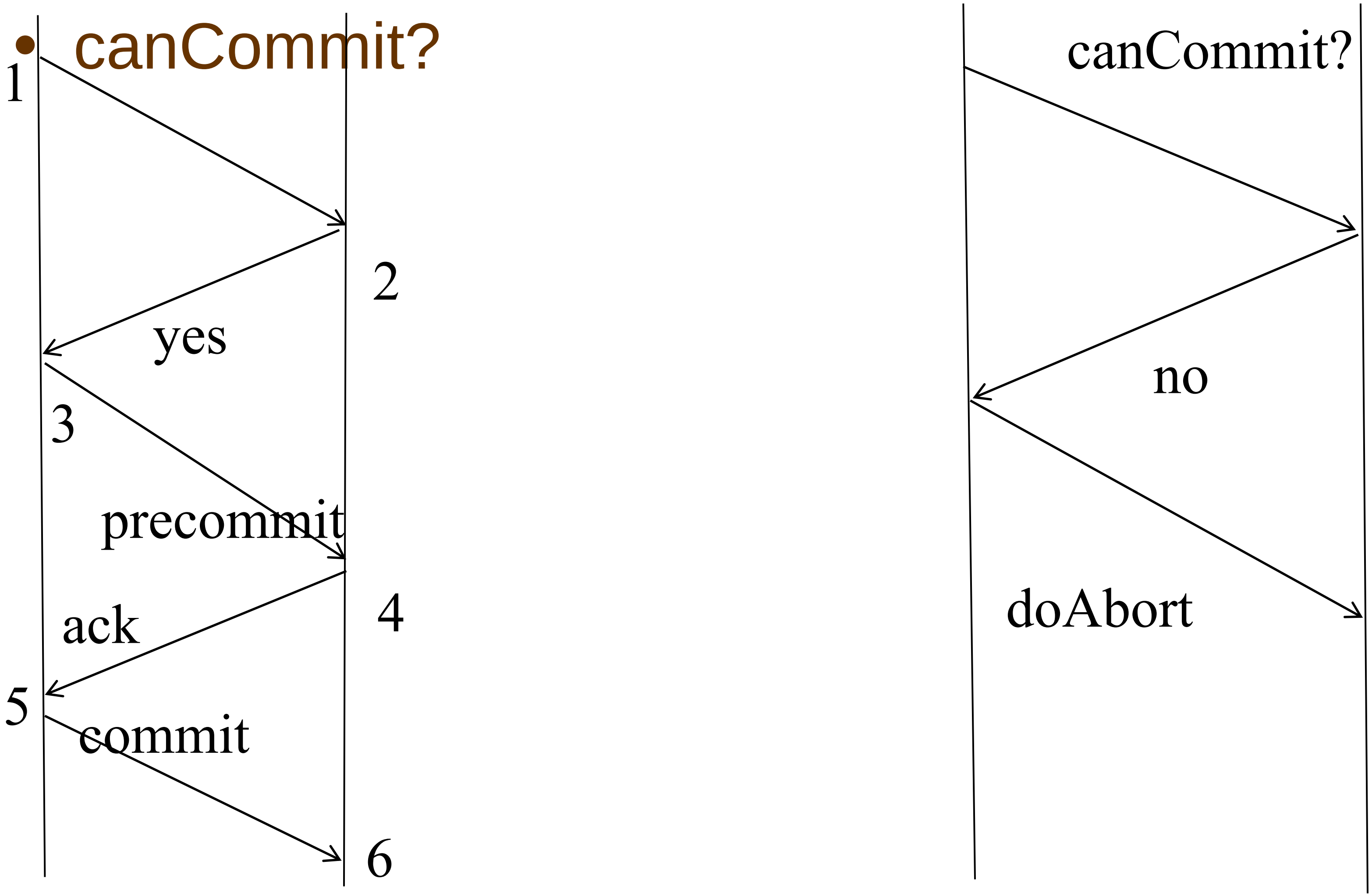
Blocking can occur if:

- Coordinatoor crashes
- Cohort cannot communicate with coordinator
- Between 2 and 4

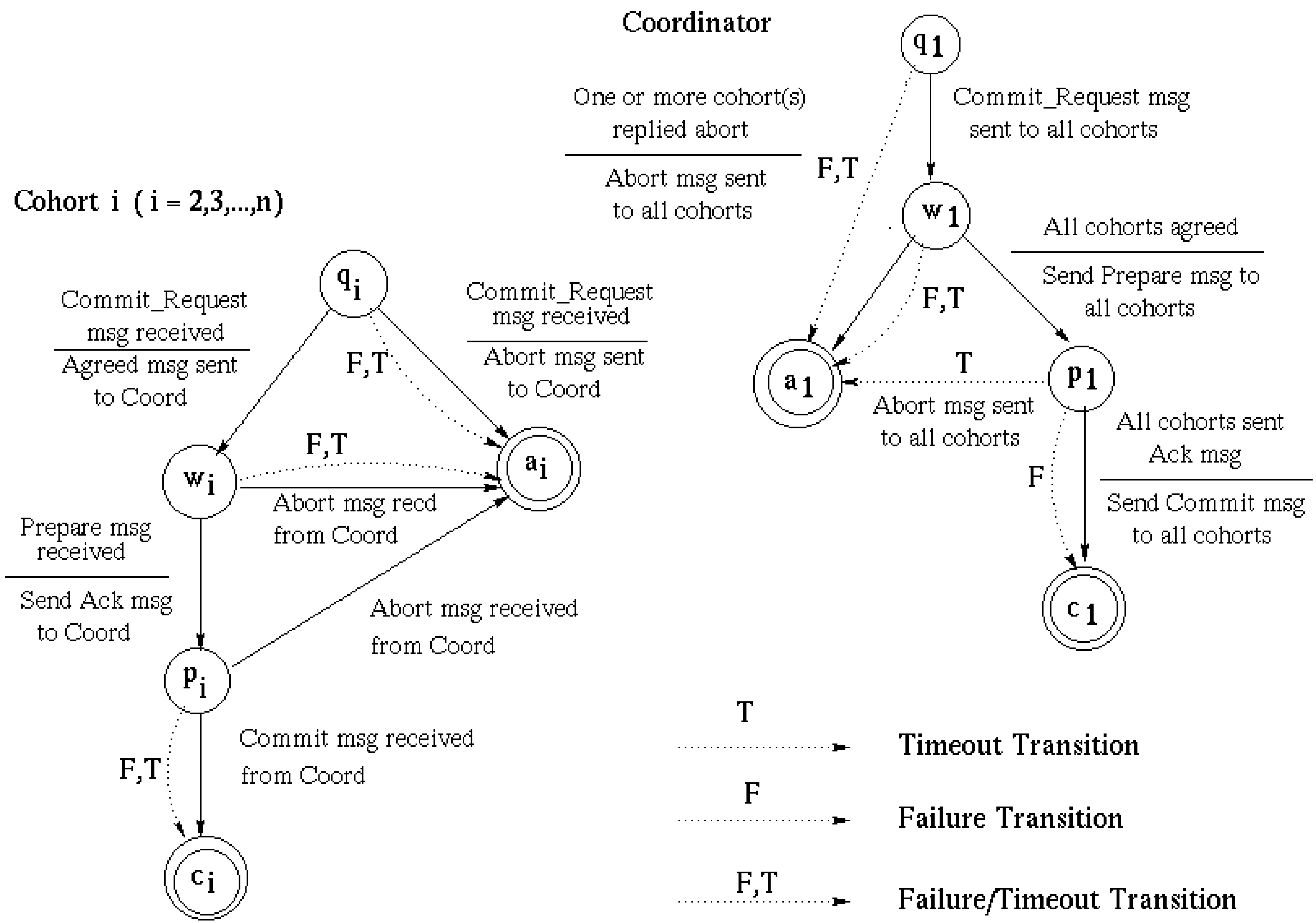


Three-Phase Commit Protocol

- canCommit?



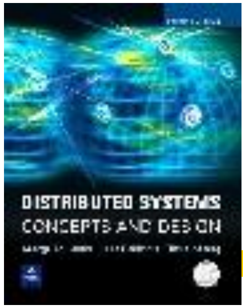
Three-phase commit protocol



Performance of the two-phase commit protocol

- if there are no failures, the 2PC involving *N* participants requires
  - *N canCommit?* messages and replies, followed by *N doCommit* messages.
    - the cost in messages is proportional to 3*N*, and the cost in time is three rounds of messages.
    - The *haveCommitted* messages are not counted
  - there may be arbitrarily many server and communication failures
  - 2PC is is guaranteed to complete eventually, but it is not possible to specify a time limit within which it will be completed
    - delays to participants in uncertain state
    - some 3PCs designed to alleviate such delays
      - they require more messages and more rounds for the normal case

Teaching material based on Distributed Systems: Concepts and Design, Edition 3, Addison-Wesley 2001.



Distributed Systems Course

Distributed transactions

Viewing: These slides must be viewed in slide show mode.

13.1 Introduction

13.2 Flat and nested distributed transactions

13.3 Atomic commit protocols

13.4 Concurrency control in distributed transactions

13.5 Distributed deadlocks

13.6 Transaction recovery

13.3.2 Two-phase commit protocol for nested transactions

- Recall Fig 13.1b, top-level transaction T and subtransactions T_1, T_2, T_{11}, T_{12}, T_{21}, T_{22}
- A subtransaction starts after its parent and finishes before it
- When a subtransaction completes, it makes an independent decision either to *commit provisionally* or to abort.
  - A provisional commit is not the same as being prepared: it is a local decision and is not backed up on permanent storage.
  - If the server crashes subsequently, its replacement will not be able to carry out a provisional commit.
- A two-phase commit protocol is needed for nested transactions
  - it allows servers of provisionally committed transactions that have crashed to abort them when they recover.