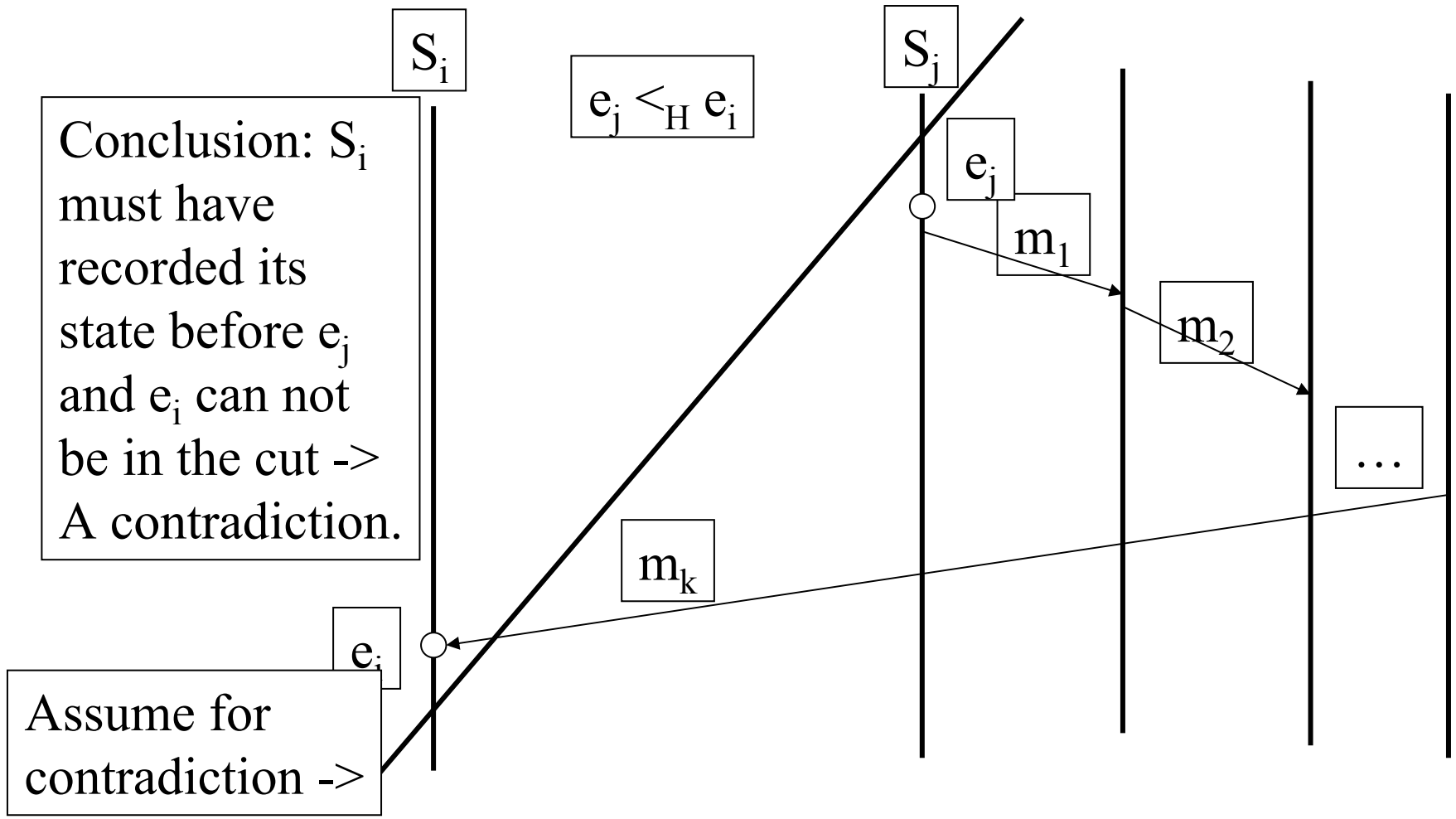
Distributed Transaction Management – 2003

Jyrki Nummenmaa
http://www.cs.uta.fi/~dtm
jyrki@cs.uta.fi

Theorem: The snapshot algorithm selects a consistent cut

- Proof. Let e_i and e_j be events, which occurred on sites S_i and S_j and $e_j <_H e_i$. Assume that e_i was in the cut produced by the snapshot algorithm. We want to show that also e_j is in the cut. If $S_i = S_j$, this is clear. Assume that $S_i \neq S_j$.
  Assume, now, for contradiction, that e_j was not in the cut. Consider the sequence of messages m_1 m_2 …m_k by which $e_j <_H e_i$. By the way snap markers are sent and received, a snap marker message has reached S_j before each m_1 m_2 …m_k and S_i has therefore recorded its state before e_j. Therefore, e_i is not in the cut. This is a contradiction, and we are done.

A picture relating to the theorem



Some observations on the example LockManager implementation

Example lock management implementation - concurrency

- A lock manager uses threads to serve incoming connections.
- As threads access the lock manager's services concurrently, those methods are qualified to be synchronized.There may actually be an overkill, as it seems to be enough to synchronize the public methods (xlock/slock/unlock). I will still have a look at that.
- An interactive client uses a single thread, but this is just to access the sleep method. There is no concurrent data access and consequently no synchronized methods.

Example lock management implementation – input/output

- Each LMMultiServerThread talks with only one client.
- Each client talks to several servers (LMMultiServerThreads).
- An interactive client talks also listens to a client controller, but it does not read standard input (this would block, or if you just check input with .ready() then user input is not echoed on the screen :(
- A client controller only sends reads standard input and sends the messages to an interactive client.

Example lock management implementation – server efficiency

- It would be more efficient to pick threads from a pool of idle threads rather than always creating a new one.
- The data structures are not optimized. As there typically is a large number of data items and only some unpredictable number of them is locked at any one moment, a hashing data structure might be suitable for the lock table (like HashTable, but a hash function must exist for the hash data type, e.g. Integer is ok).
- It would be ok to connect a lock request queue (e.g. a linked list) for each lock item in the lock table with a direct reference.

Example lock management implementation – txn ids

- Currently integers are used.
- Giving global txn ids sequentially (over all sites) is either complicated or needs a centralised server -> it is better to use a local (site-id, local-txn-id) pair to construct the txn ids. If the sites are known in advance, they can be numbered.
- We only care about txns accessing some resource manager's services and there may be a fixed set of them -> ask the first server to talk with for an id (server-id, local-txn-id).

How distributed txns are born?

- Distributed database management systems: a client contacts local server, which takes care of transparent distribution – the client does not need to know about the other sites.
- Alternative: a client contacts all local resource managers directly.
- Alternative: separate client programs are started on all respective sites.
- Typically the client executing on the first site acts as a coordinator communicating with the other clients.

Distributed Commit

Ending the transaction

- Finally, a txn is to either commit, in which case its updates are made permanent on all sites, or the txn is rolled back, in which case none of its updates are made permanent.
- The servers must agree on the fate of the transaction. For this, the servers negotiate (vote).
- The local servers participating in the commit process are called *participants*.

Failure model – servers / 1

- In our work on atomic commitment protocols, we make the assumption that a server is either working correctly or not at all.
- In our model, a server never performs incorrect actions. Of course, in reality servers do produce incorrect behaviour.
- A *partial failure* is a situation where some servers are operational while others are down.
- Partial failures can be tricky because operational servers may be uncertain about the status of failed servers.
- The operational servers may become blocked, i.e. unable to commit or roll back a txn until such uncertainty is resolved.

Failure model – servers / 1

- The reason why partial failures are so tricky to handle is that operational servers may be uncertain about the state of the servers on failed sites. An operational server may become blocked, unable to commit or rollback the txn. Thus it will have to hold locks on the data items that have been accessed by the txn, thereby preventing other txns from progressing.
- Not surprisingly, an important design goal for atomic commitment protocols is to minimise the effect of one site's failure on another site's ability to continue processing.
- A crashed site may recover.

Failure model – network

- Messages may get lost.
- Messages may be delayed.
- Messages are not changed.
- Communication between two sites may not be possible for some time. This is called network partitioning and this assumption is sometimes relaxed, since it creates hard problems (more on this later).
- The network partitioning may get fixed.

Detecting Failures by Timeouts

- We assume that the only way that server A can discover that it cannot communicate with server B is through the use of timeouts.
- A sends a message to B and waits for a reply within the timeout period.
- If a reply arrives then clearly A and B can communicate.
- If the timeout period elapses and A has not received a reply then A concludes that it cannot communicate with B.

Detecting Failures by Timeouts

- Choosing a reasonable value for the timeout period can be tricky.
- The actual time that it takes to communicate will depend on many hard-to-quantify variables, including the physical characteristics of the servers and the communication lines, the system load and the message routing techniques.

An atomic commit protocol

- An atomic commit protocol (`ACP') ensures consistent termination even in the presence of partial failures.
- It is not enough if the coordinator just tells the other participants to commit.
- For example, one of the participants may decide to roll back the txn.
- This fact must be communicated to the other participants because they, too, must roll back.

An atomic commit protocol

- An atomic commit protocol is an algorithm for the coordinator and participants such that either the coordinator and all of the participants commit the txn or they all roll it back.
- The coordinator has to find out if the participants can all commit the txn. It asks the participants to vote on this.
- The ACP should have the following properties.

Requirements for distributed commit (1-4)

- (1) A participant can vote **Yes** or **No** and may not change the vote.
- (2) A participant can decide either *Abort* or *Commit* and may not change it.
- (3) If any participant votes **No**, then the global decision must be *Abort*.
- (4) It must never happen that one participant decides *Abort* and another decides *Commit*.

Requirements for distributed commit (5-6)

- (5) All participants, which execute sufficiently long, must eventually decide, regardless whether they have failed earlier or not.
- (6) If there are no failures or suspected failures and all participants vote **Yes**, then the decision must not be *Abort*.

  The participants are not allowed to create artificial failures or suspicion.

Observations

- Having voted **No**, a participant may unilaterally rollback. (One **No** vote implies global rollback.)
- However, having voted **Yes**, a participant may not unilaterally commit. (Somebody else may have voted **No** and then rolled back.)
- Note that it is possible that all participants vote to Commit, and yet the decision is to Rollback.

Observations

- Note that condition AC1 does not require that all participants reach a decision: some servers may fail and never recover.
- We do not even require that all participating servers that remain operational reach a decision.
- However, we do require that all participating servers be able to reach a decision once failures are repaired

Uncertainty period

- The period between voting to commit and receiving information about the overall decision is called the *uncertainty period* for the participating server.
- During that period, we say that the participant is uncertain.
- Suppose that a failure in the interconnection network disables communication between a participant, P, and all other participants and the coordinator, while P is uncertain. Then P cannot reach a decision until after the network failure has been repaired.

Distributed Two-Phase Commit Protocol (2PC)

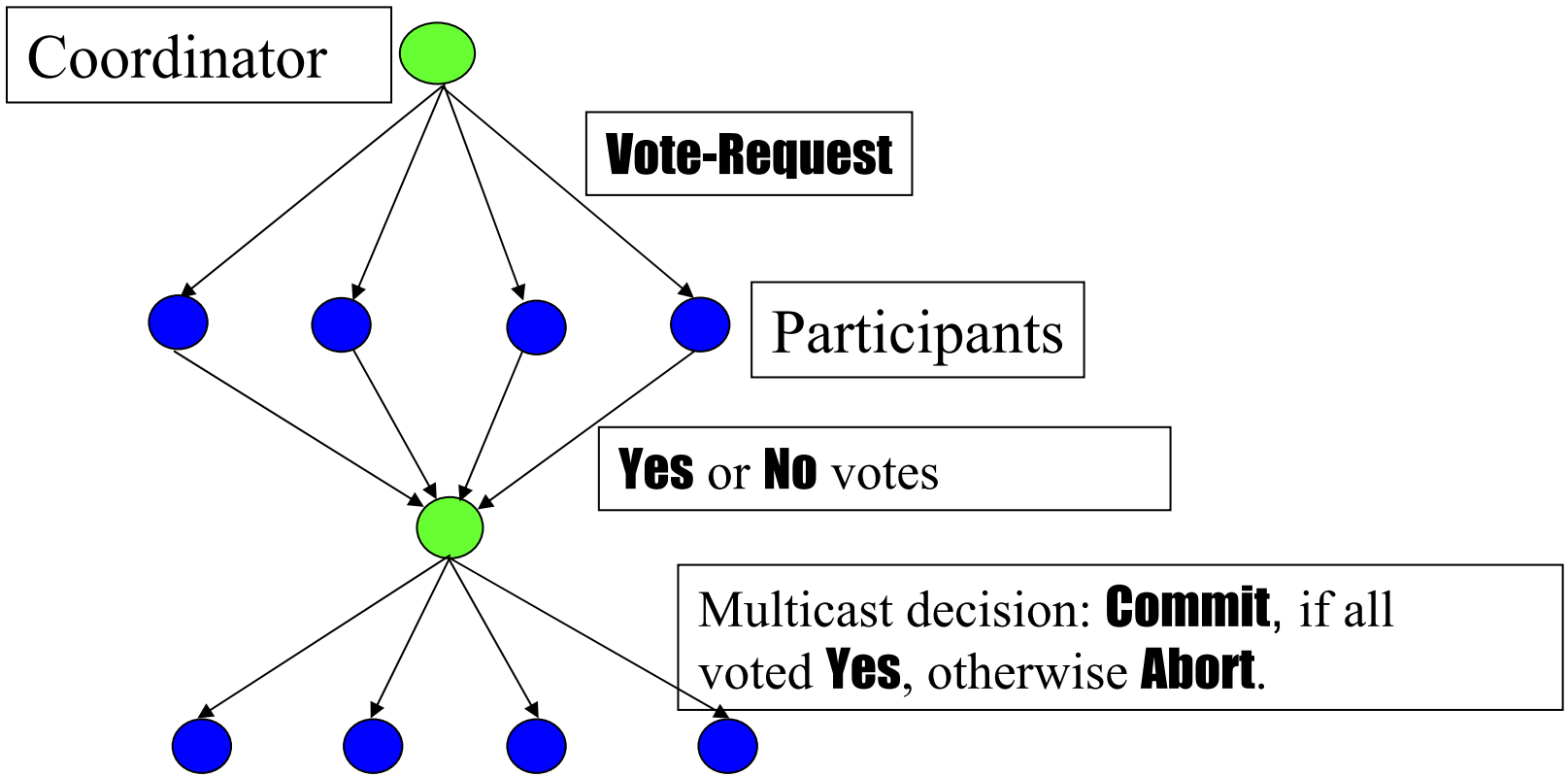
Basic 2PC

- Many DBMS products use the two-phase commit protocol (2PC) as the process by which txns that use multiple Servers commit in a consistent manner.
- Typically, the majority of txns in an application do not use multiple servers. Such txns are unaffected by 2PC.
- Txns that write-access data on multiple servers commit in two phases: a voting phase and a decision phase.

Basic 2PC

- First, all of the servers vote on the txn, indicating whether they are able to commit or not.
- A server may vote **No** if, for example, it has run out of disk space or if it must pre-emptively rollback the txn to break a deadlock.
- If all of the votes are **Yes,** the decision is made to commit, and all of the servers are told to commit.

2PC for Distributed Commit



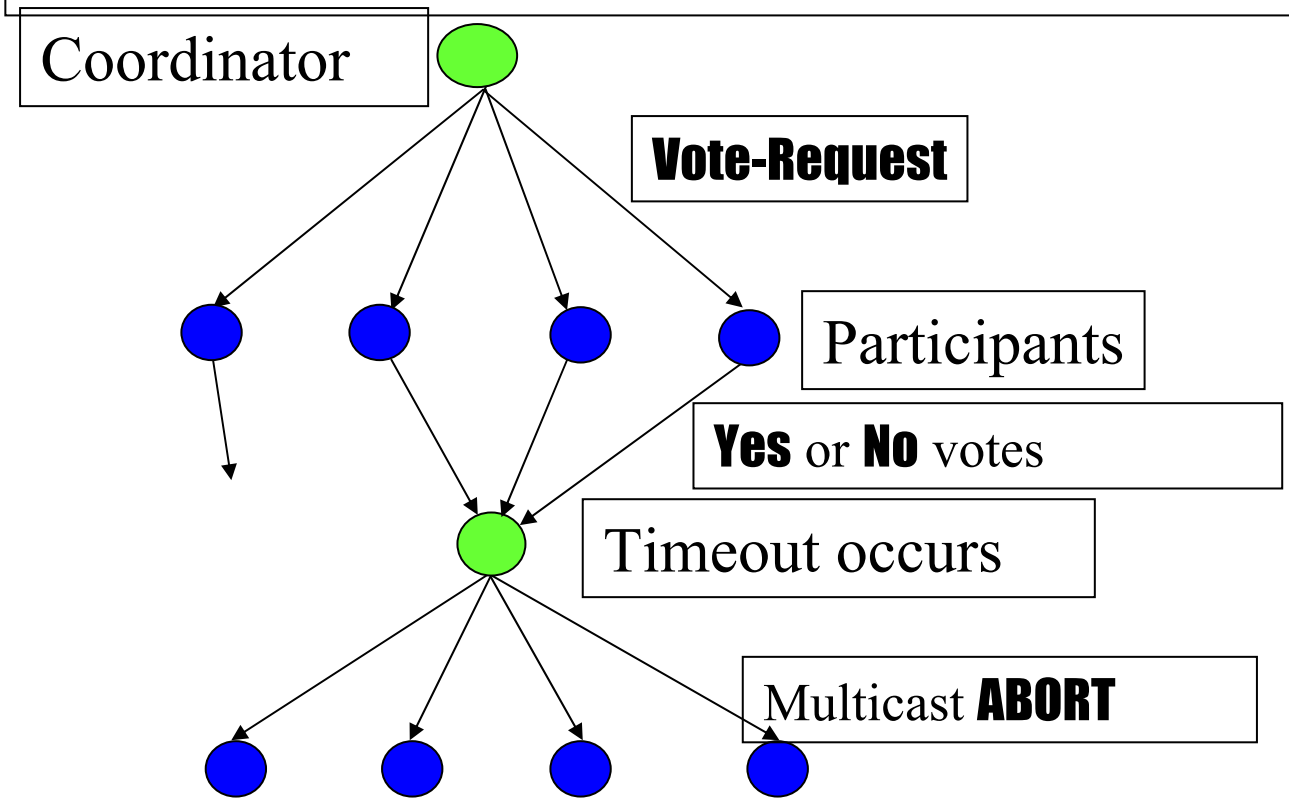
2PC Coordinator

- Initiate voting by sending **Vote-Req** to all participants and enters a wait-state.
- Having received one **No** or a timeout in wait-state, decides *Abort* and sends the decision to all participants.
- Having received a **Yes** from all in wait-state (before timing out), decides *Commit* and sends the decision to all participants.

2PC Participant

- Having received a **Vote-Req**, either
  - sends a **No** and decides *Rollback* locally
  - sends a **Yes** and enters a wait-state.
- If in wait-state receives either *Rollback* or *Commit*, decides accordingly. If it times out in wait-state, it concludes that the coordinator is down and starts a termination protocol – more on this soon.

2PC - a timeout occurs



Centralised vs. Decentralised

- There is also a version of 2PC whereby there is no coordinator and the servers all send messages to each other. This is called decentralised 2PC.
- 2PC with a coordinator is called centralised 2PC.
- We shall exemplify their behaviour with a simulation applet at http://www.cs.uta.fi/utacsoft/discosim

Timeout actions

- We assume that the participants and the coordinator try to detect failures with timeouts.
- If a participant times out before Vote-Req, it can decide to Rollback.
- If the coordinator times out before sending out the outcome of the vote to anyone, it can decide to rollback.
- If a participant times out after voting Yes, it must use a *termination protocol*.

Termination protocol

- If a participant times out having voted Yes, it must consult the others about the outcome. Say, P is consulting Q.
- If Q has received the decision from the coordinator, it tells it to P.
- If Q has not yet voted, it can decide to Rollback.
- If Q does not know, it can’t help.
- If no-one can help P, P is blocked.

Recovery

- When a crashed participant recovers, it checks its log.
- P can recover independently, if P recovers
  - having received the outcome from the coordinator,
  - not having voted, or
  - having decided to rollback.
- Otherwise, P must consult the others, similarly as in the termination protocol.
- *Note: the simulation applet does not contain termination and recovery!*

2PC Analysis

- (1) A participant can vote **Yes** or **No** and may not change the vote.
  - This should be clear.
- (2) A participant can decide either *Abort* or *Commit* and may not change it.
  - This should also be clear.

2PC Analysis

- (3) If any participant votes **No**, then the global decision must be *Abort*.
  - Note that the global decision is *Commit* only when everyone votes **Yes**, and since no-one votes both **Yes** and **No**, this should be clear.
- (4) It must never happen that one participant decides *Abort* and another decides *Commit*.
  - On similar lines than (3).

2PC Analysis

- (5) All participants, which execute sufficiently long, must eventually decide, regardless whether they have failed earlier or not.
  - Only holds, if we assume that failed sites always recover and they do it for sufficiently long so that the 2PC recovery can make progress.
- (6) If there are no failures or suspected failures and all participants vote **Yes**, then the decision must not be *Abort*. The participants are not allowed to create artificial failures or suspicion.
  - Ok.

2PC Analysis

- Sometimes there is intereste towards the number of messages exchanged. In the simplest case, the 2PC involves 3N messages.
- Recovery and termination change these figures.
- However, the assumption is that most txns terminate normally.

What does the coordinator write to a log?

- When the coordinator sends *Vote-req*, it writes a *start-2PC* record and a list containing the identities of the participants to its log. It also sends this list to each participant at the same time as the *Vote-req* message.
- Before the coordinator sends Commit to the participants, it writes a *commit* record in the log.
- If the coordinator sends *rollback* to the participants, it writes a *rollback* to the log.

What does a participant write to a log?

- If a participant votes Yes, it writes a *yes* record to its log *before* sending *yes* to the coordinator. This log record contains the name of the coordinator and the list of the participants.
- If this participant votes No, it writes a *rollback* record to the log and then sends the No vote to the coordinator.
- After receiving Commit (or Rollback, a participant writes a *commit* (or a *rollback* record into the log.

How to recover using the log?

- If the log of S contains a *start-2PC,* record then S was the host of the coordinator. If it also contains a *commit* or *rollback* record, then the coordinator had decided before the failure occurred. If neither record is found in the log then the coordinator can now unilaterally decide to Rollback by inserting a *rollback* record in the log.

How to recover using the log / 2

- If the log does not contain a start-2PC record, then S was the host of a participant. There are three cases to consider:
- (1) The log contains a *commit* or *rollback* record. Then the participant had reached its decision before failure.
- (2) The log does not contain a *yes* record. Then either the participant failed before voting or voted No (but did not write a *rollback* record before failing.) (This is why the Yes record must be written before Yes is sent.) -> Rollback

How to recover using the log / 3

- (3) The log contains a yes but not *commit* or *rollback* record. Then the participant failed while in this uncertainty period.
- It can try to reach a decision using the termination protocol.
- A *yes* record includes the names of the coordinator and participants; these are needed for the termination protocol.

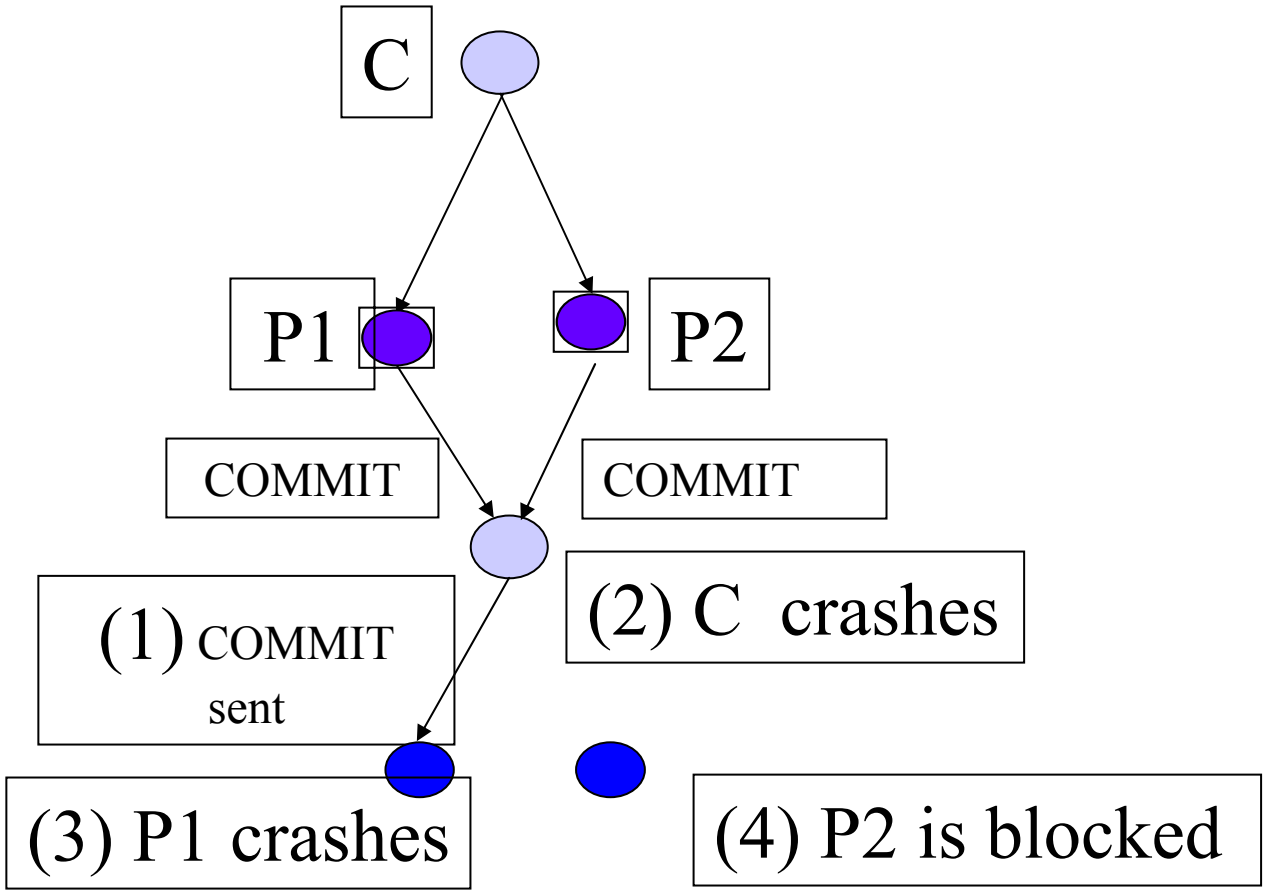
Garbage collection

- Eventually, it will become necessary to garbage collect the space in the log. There are two principles to adopt:
- [GC1:] A site cannot delete log records of a txn, T, from its log until at least after the resource manager has processed the commit or the rollback for T.
- [GC2:] At least one site must not delete the records of T from its log until that site has received messages indicating that the resource managers at all other sites have processed the commit or rollback of T.

Blocking

- We say that a participant is *blocked,* if it must await the repair of a fault before it is able to proceed.
- A txn may remain unterminated, holding locks, for arbitrarily long periods of time at the blocked server.
- Suppose that participant, S, fails whilst it is uncertain. When S recovers, it cannot reach a decision on its own. It must communicate with the other participants to find out what was decided.
- We say that a commitment protocol is *non-blocking* if it permits termination without waiting for the recovery of failed servers.

Two-Phase Commit Blocks



Blocking Theorem

- Theorem 1: If it is possible for the network to partition, then any Atomic Commit Protocol may cause processes to become blocked.
- Sketch of proof. Assume two sites, A and B, and a protocol. Assume that Kth message M is the one after which commit decision can be made. Assume that A can commit having sent M to B, but not before.

  If network partitions just before the Kth message, B is blocked. (Does not know, if A sent it and committed, or did not send it, in which case it might have decided abort.)