# 7. Two Phase Commit

CSEP 545 Transaction Processing
for E-Commerce

Philip A. Bernstein

# Outline

1. Introduction
2. The Two-Phase Commit (2PC) Protocol
3. 2PC Failure Handling
4. 2PC Optimizations
5. Process Structuring
6. Three Phase Commit

# 7.1 Introduction

- Goal - ensure the atomicity of a transaction that accesses multiple resource managers
- (Recall, resource abstracts data, messages, and other items that are shared by transactions.)
- Why is this hard?
  - What if resource manager $RM_i$ fails after a transaction commits at $RM_k$?
  - What if other resource managers are down when $RM_i$ recovers?
  - What if a transaction thinks a resource manager failed and therefore aborted, when it actually is still running?

# Assumptions

- Each resource manager independently commits or aborts a transaction atomically on its resources.
- Home(T) decides when to start committing T
- Home(T) doesn't start committing T until T terminates at all nodes (possibly hard)
- Resource managers fail by stopping
  - no Byzantine failures, where a failed process exhibits arbitrary behavior, such as sending the wrong message

# Problem Statement

- Transaction T accessed data at resource managers $R_1$, … $R_n$.
- The goal is to either
  - commit T at all of $R_1$, … $R_n$, or
  - abort T at all of $R_1$, … $R_n$
  - even if resource managers, nodes and communications links fail during the commit or abort activity
- That is, never commit at $R_i$ but abort at $R_k$.

# 7.2 Two-Phase Commit

- Two phase commit (2PC) is the standard protocol for making commit and abort atomic
- Coordinator - the component that coordinates commitment at home(T)
- Participant - a resource manager accessed by T
- A participant P is ready to commit T if all of T's after-images at P are in stable storage
- The coordinator must not commit T until all participants are ready
  - If P isn't ready, T commits, and P fails, then P can't commit when it recovers.

# The Protocol

1 (Begin Phase 1) The coordinator sends a **Request-to-Prepare** message to each participant

2 The coordinator waits for all participants to vote

3 Each participant

- votes **Prepared** if it's ready to commit
- may vote **No** for any reason
- may delay voting indefinitely

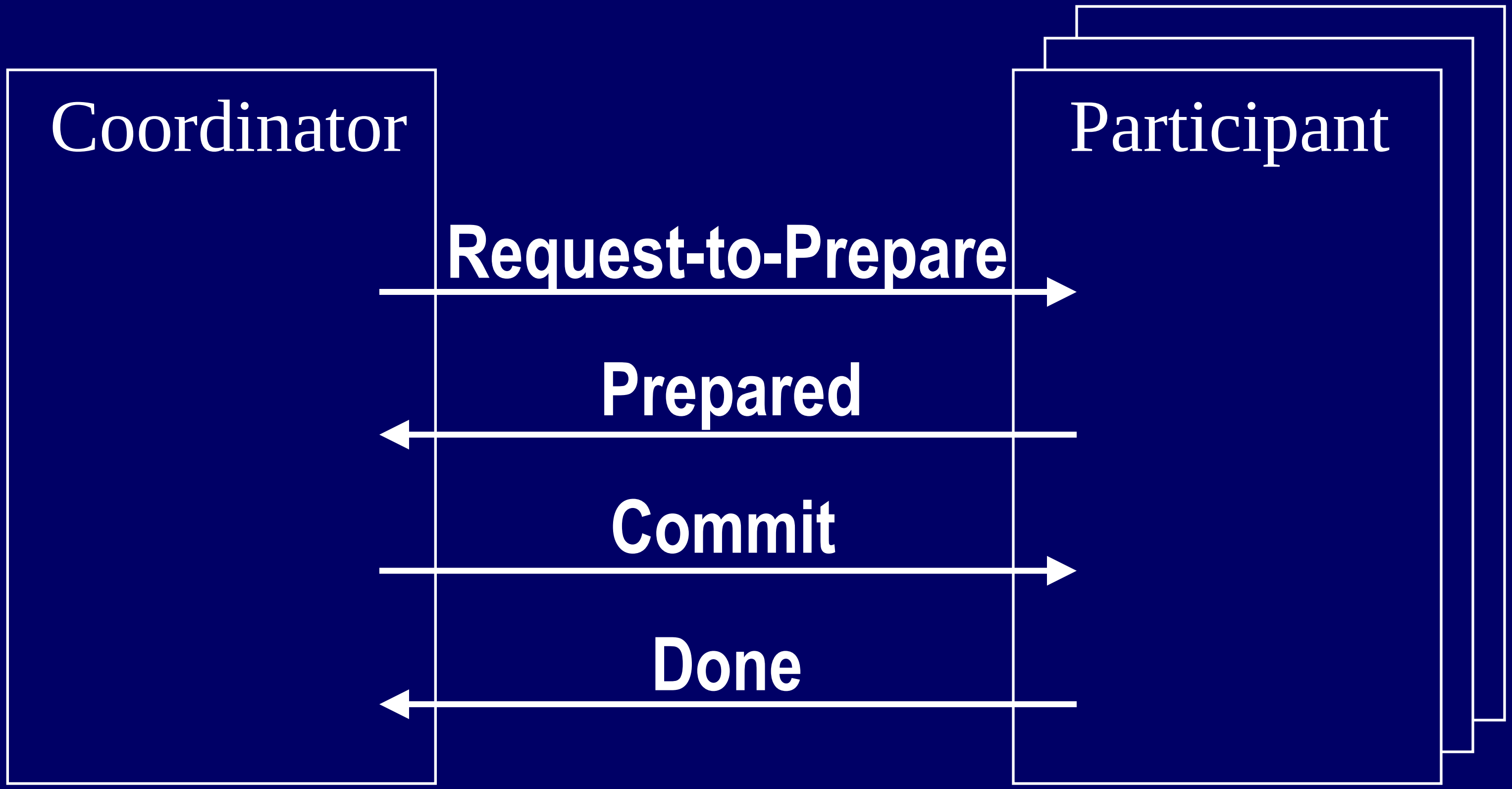
4 (Begin Phase 2) If coordinator receives **Prepared** from <u>all</u> participants, it decides to commit. (The transaction is now committed.) Otherwise, it decides to abort.

# The Protocol (cont'd)

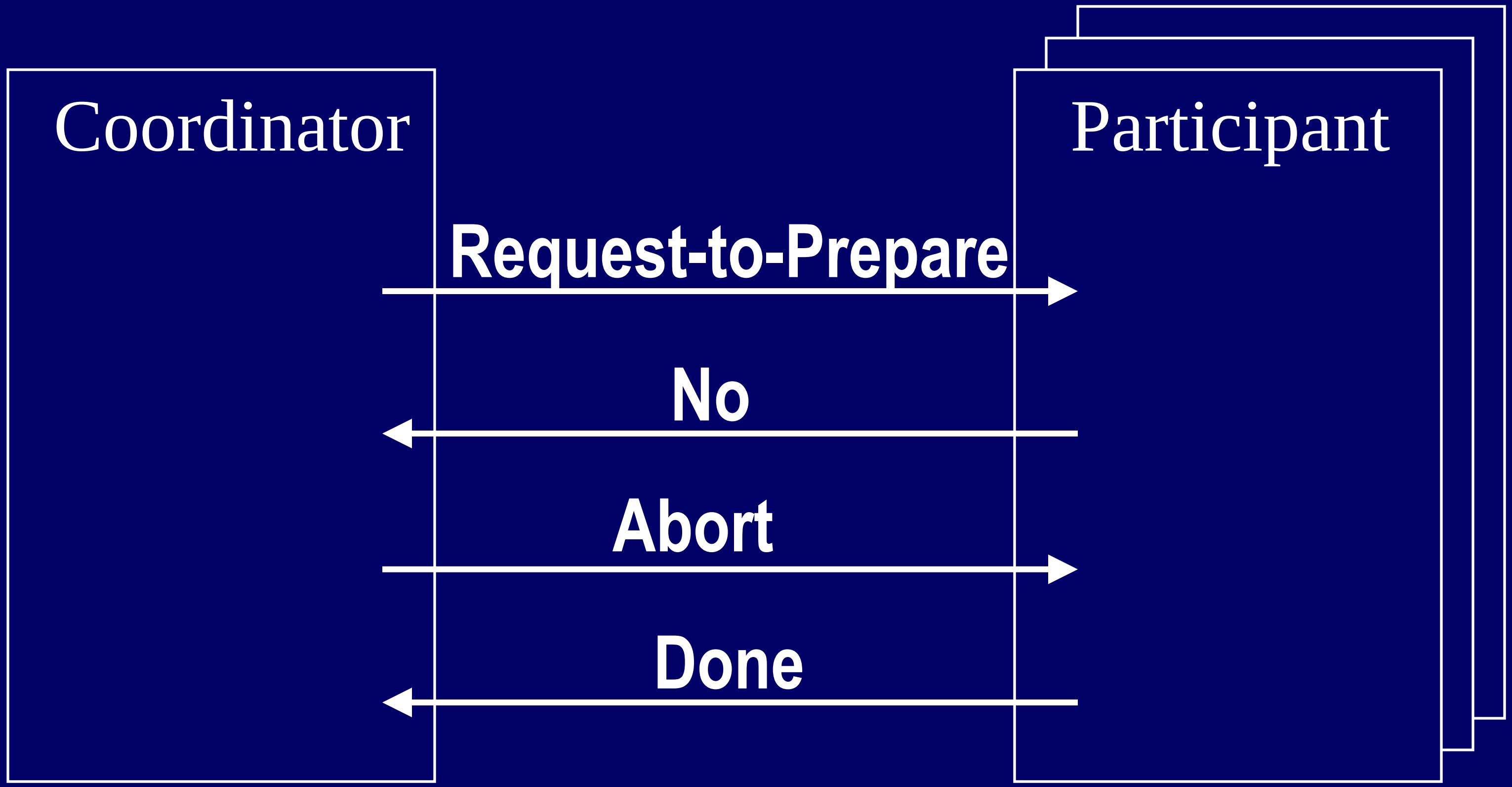
5 The coordinator sends its decision to all participants (i.e., **Commit** or **Abort**)

6 Participants acknowledge receipt of **Commit** or **Abort** by replying **Done**.

# Case 1: Commit



# Case 2: Abort

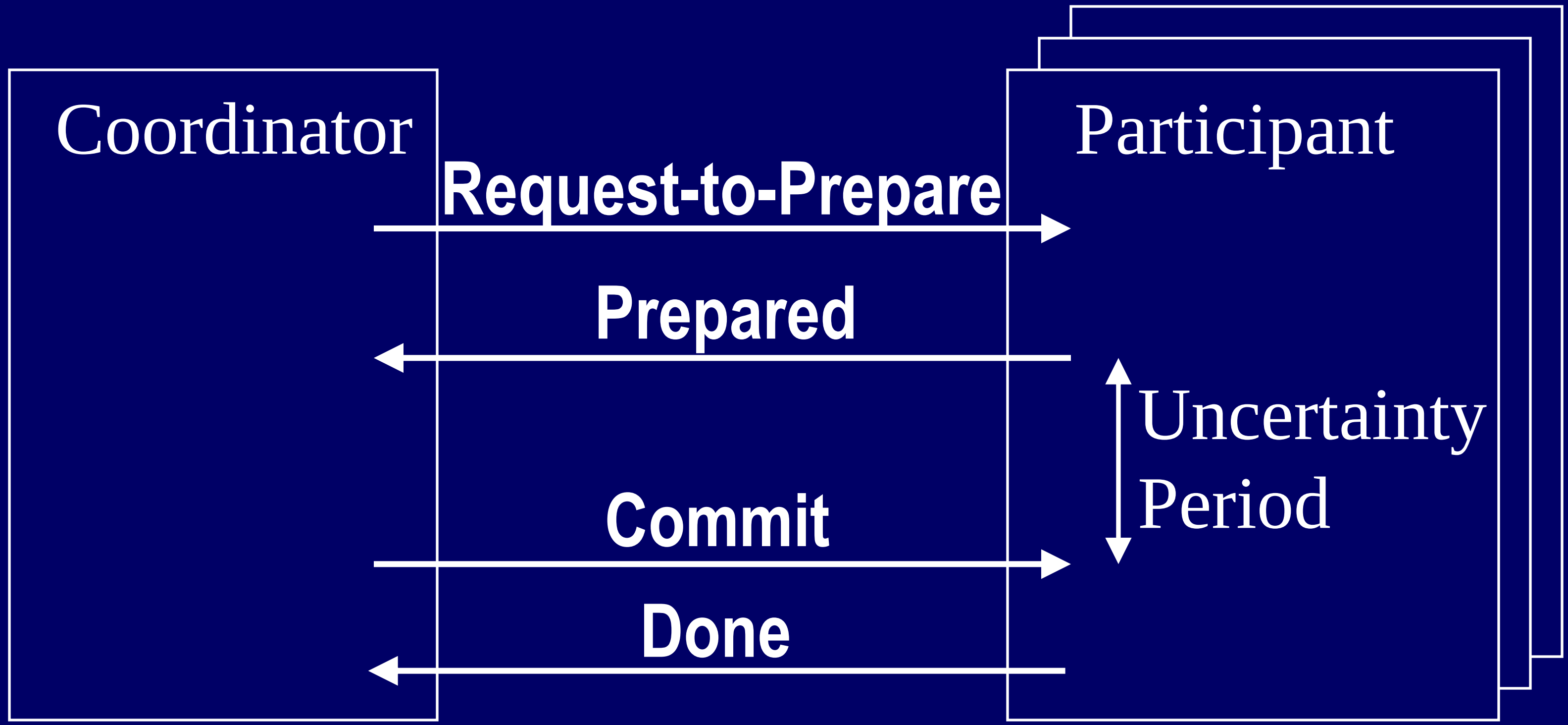


# Performance

- In the absence of failures, 2PC requires 3 rounds of messages before the decision is made known to RM's.
  - **Request-to-prepare**
  - Votes (**Prepared, No)**
  - Decision **(Commit, Abort)**
- **Done** messages are just for bookkeeping
  - they don't affect response time
  - they can be batched

# Uncertainty

- Before it votes, a participant can abort unilaterally
- After a participant votes **Prepared** and before it receives the coordinator's decision, it is uncertain. It can't unilaterally commit or abort during its uncertainty period.



# Uncertainty (cont'd)

- The coordinator is never uncertain
- If a participant fails or is disconnected from the coordinator while it's uncertain, at recovery it must find out the decision

# The Bad News Theorems

- Uncertainty periods are unavoidable
- Blocking - a participant must await a repair before continuing. Blocking is bad.
- Theorem 1 - For every possible commit protocol (not just 2PC), a communications failure can cause a participant to become blocked.
- Independent recovery - a recovered participant can decide to commit or abort without communicating with other nodes
- Theorem 2 - No commit protocol can guarantee independent recovery of failed participants

# 7.3 2PC Failure Handling

- Failure handling - what to do if the coordinator or a participant times out waiting for a message.
  - Remember, all failures are detected by timeout
- A participant times out waiting for coordinator's **Request-to-prepare**.
  - It decides to abort.
- The coordinator times out waiting for a participant's vote
  - It decides to abort

# 2PC Failure Handling (cont'd)

- A participant that voted **Prepared** times out waiting for the coordinator's decision
  - It's blocked.
  - Use a termination protocol to decide what to do.
  - Naïve termination protocol - wait till the coordinator recovers
- The coordinator times out waiting for **Done**
  - it must resolicit them, so it can <u>forget</u> the decision

# Forgetting Transactions

- After a participant receives the decision, it may forget the transaction
- After the coordinator receives **Done** from all participants, it may forget the transaction
- A participant must not reply **Done** until its commit or abort log record is stable
  - Else, if it fails, then recovers, then asks the coordinator for a decision, the coordinator may not know

# Logging 2PC State Changes

- Logging may be <u>eager</u>
  - meaning it’s flushed to disk before the next Send Message
- Or it may be <u>lazy</u> = not eager

| Coordinator | Message | Participant |
|---|---|---|
| Log Start2PC (eager) | Request-to-Prepare → | |
| | ← Prepared | Log prepared (eager) |
| Log commit (eager) | Commit → | |
| | ← Done | Log commit (eager) |
| Log commit (lazy) | | |

# Coordinator Recovery

- If the coordinator fails and later recovers, it must know the decision. It must therefore log
  - the fact that it began T's 2PC protocol, including the list of participants, and
  - Commit or Abort, before sending **Commit** or **Abort** to any participant (so it knows whether to commit or abort after it recovers).
- If the coordinator fails and recovers, it resends the decision to participants from which it doesn't remember getting **Done**
  - If the participant forgot the transaction, it replies **Done**
  - The coordinator should therefore log Done after it has received them all.

# Participant Recovery

- If a participant P fails and later recovers, it first performs centralized recovery (Restart)
- For each distributed transaction T that was active at the time of failure
  - If P is not uncertain about T, then it unilaterally aborts T
  - If P is uncertain, it runs the termination protocol (which may leave P blocked)
- To ensure it can tell whether it's uncertain, P must log its vote <u>before</u> sending it to the coordinator
- To avoid becoming totally blocked due to one blocked transaction, P should reacquire T's locks during Restart and allow Restart to finish before T is resolved.

# Heuristic Commit

- Suppose a participant recovers, but the termination protocol leaves T blocked.
- Operator can guess whether to commit or abort
  – Must detect wrong guesses when coordinator recovers
  – Must run compensations for wrong guesses
- Heuristic commit
  – If T is blocked, the local resource manager (actually, transaction manager) guesses
  – At coordinator recovery, the transaction managers jointly detect wrong guesses.

# 7.4 2PC Optimizations and Variations

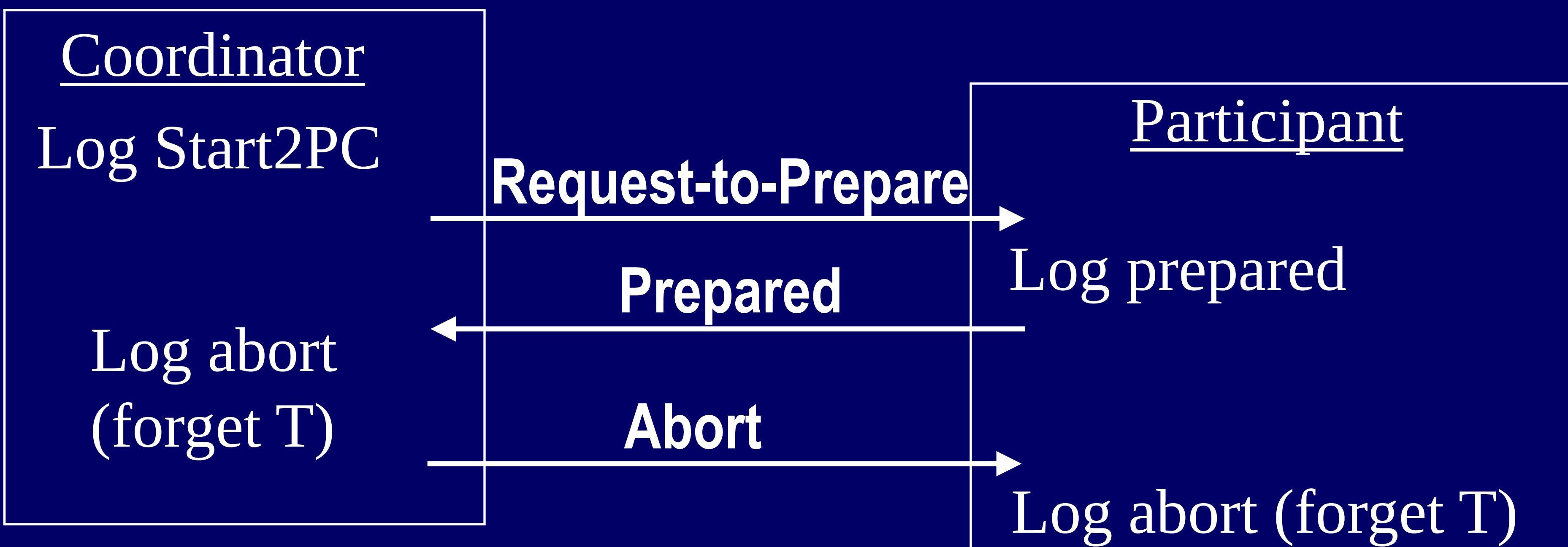
- Optimizations
  - Read-only transaction
  - Presumed Abort
  - Transfer of coordination
  - Cooperative termination protocol
- Variations
  - Re-infection
  - Phase Zero

# Read-only Transaction

- A read-only participant need only respond to phase one. It doesn’t care what the decision is.
- It responds **Prepared-Read-Only** to **Request-to-Prepare**, to tell the coordinator not to send the decision
- Limitation - All other participants must be fully terminated, since the read-only participant will release locks after voting.
  – No more testing of SQL integrity constraints
  – No more evaluation of SQL triggers

# Presumed Abort

- After a coordinator decides Abort and sends **Abort** to participants, it forgets about T immediately.
- Participants don't acknowledge **Abort** (with **Done**)

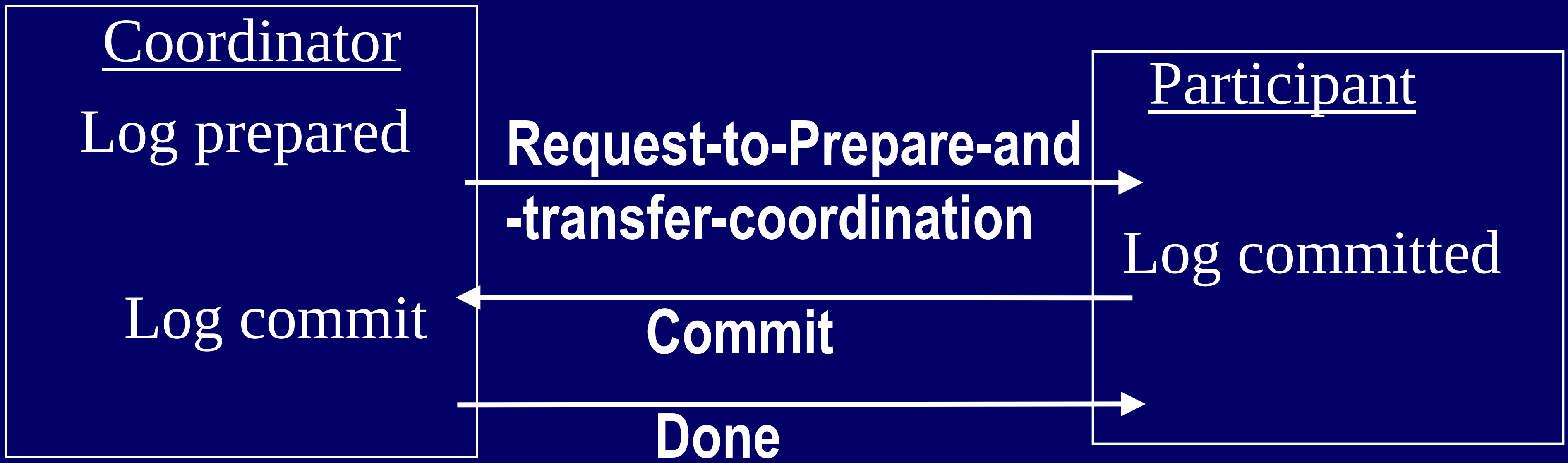



- If a participant times out waiting for the decision, it asks the coordinator to retry.
  - If the coordinator has no info for T, it replies **Abort**.

# Transfer of Coordination

If there is one participant, you can save a round of messages

1. Coordinator asks participant to prepare and become the coordinator.
2. The participant (now coordinator) prepares, commits, and tells the former coordinator to commit.
3. The coordinator commits and replies Done.




- Supported by some app servers, but not in any standards.

# Cooperative Termination Protocol (CTP)

- Assume coordinator includes a list of participants in **Request-to-Prepare**.
- If a participant times-out waiting for the decision, it runs the following protocol.

1. Participant P sends **Decision-Req** to other participants
2. If participant Q voted **No** or hasn't voted or received **Abort** from the coordinator, it responds **Abort**
3. If participant Q received **Commit** from the coordinator, it responds **Commit**.
4. If participant Q is uncertain, it responds **Uncertain** (or doesn't respond at all).

- If all participants are uncertain, then P remains blocked.

# Cooperative Termination Issues

- Participants don't know when to forget T, since other participants may require CTP
  - Solution 1 - After receiving **Done** from all participants, coordinator sends **End** to all participants
  - Solution 2 - After receiving a decision, a participant may forget T any time.
- To ensure it can run CTP, a participant should include the list of participants in the vote log record.

# Reinfection

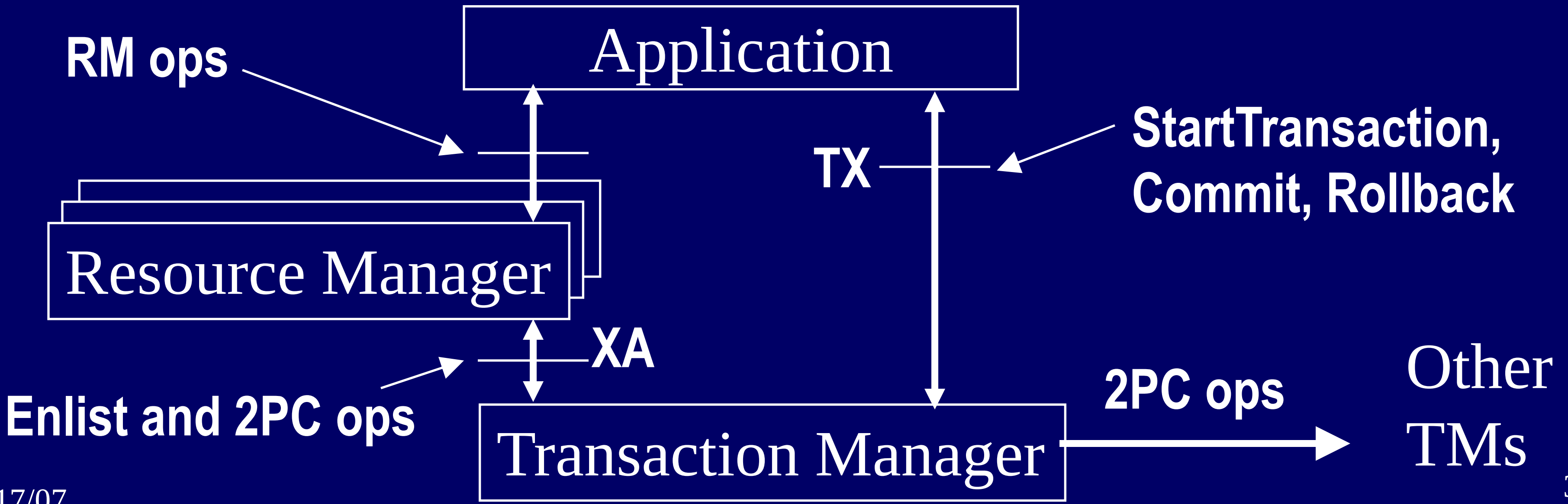
- Suppose A is coordinator and B and C are participants
  - A asks B and C to prepare
  - B votes prepared
  - C calls B to do some work. (B is reinfected.)
  - B does the work and tells C it has prepared, but now it expects C to be its coordinator.
  - When A asks C to prepare, C propagates the request to B and votes prepared only if both B and C are prepared. (See Tree of Processes discussion later.)
- Can be used to implement integrity constraint checking, triggers, and other commit-time processing, without requiring an extra phase (between phases 1 and 2 of 2PC).

# Phase Zero

- Suppose a participant P is caching transaction T's updates that P needs to send to an RM (another participant) before T commits.
  – P must send the updates after T invokes Commit, to ensure P has all of T's updates
  – P must send the updates before the RM prepares, to ensure the updates are made stable during phase one.
  – Thus, we need an extra phase, before phase 1.
- A participant explicitly enlists for phase zero.
  – It doesn't ack phase zero until it finishes flushing its cached updates to other participants.
- Supported in Microsoft DTC.

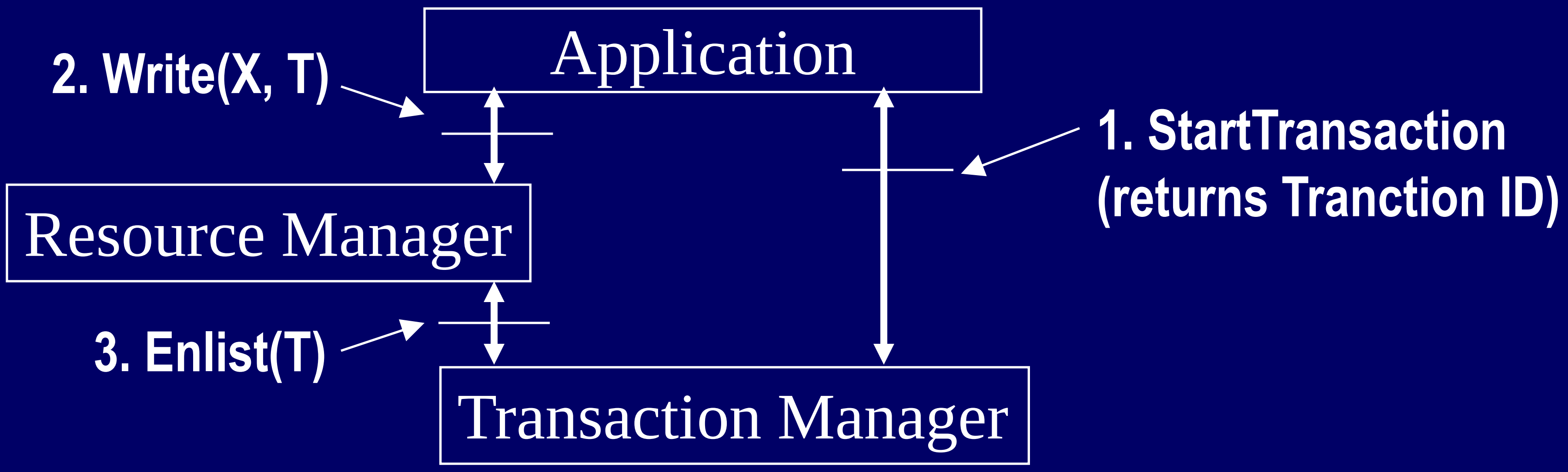
# 7.5 Process Structuring

- To support multiple RMs on multiple nodes, and minimize communication, use one transaction manager (TM) per node
- TM may be in the OS (VAX/VMS, Win), the app server (IBM CICS), DBMS, or a separate product (early Tandem).
- TM performs coordinator and participant roles for all transactions at its node.
- TM communicates with local RMs and remote TMs.



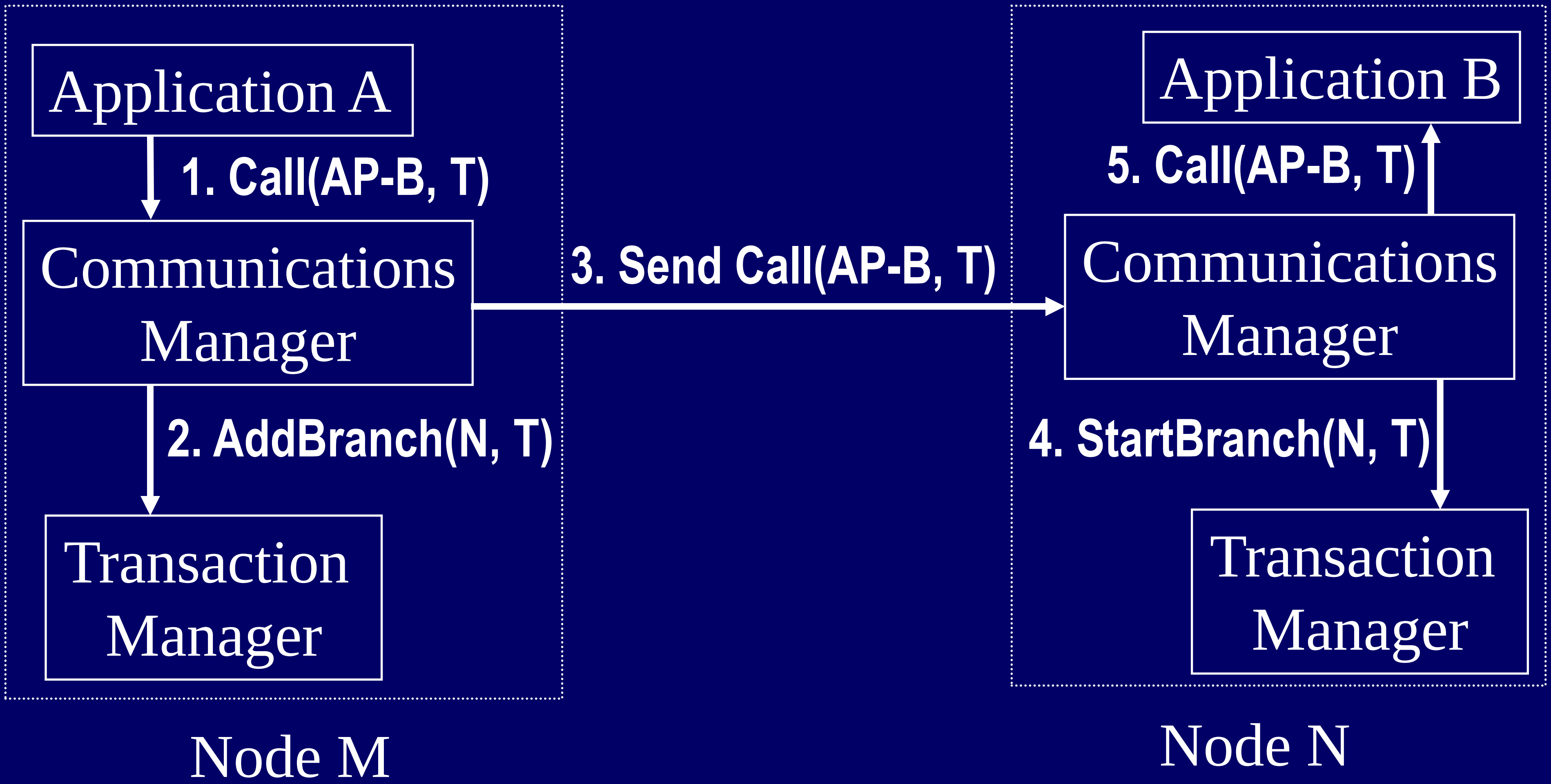
# Enlisting in a Transaction

- When an Application in a transaction T first calls an RM, the RM must tell the TM it is part of T.
- Called <u>enlisting</u> or <u>joining</u> the transaction



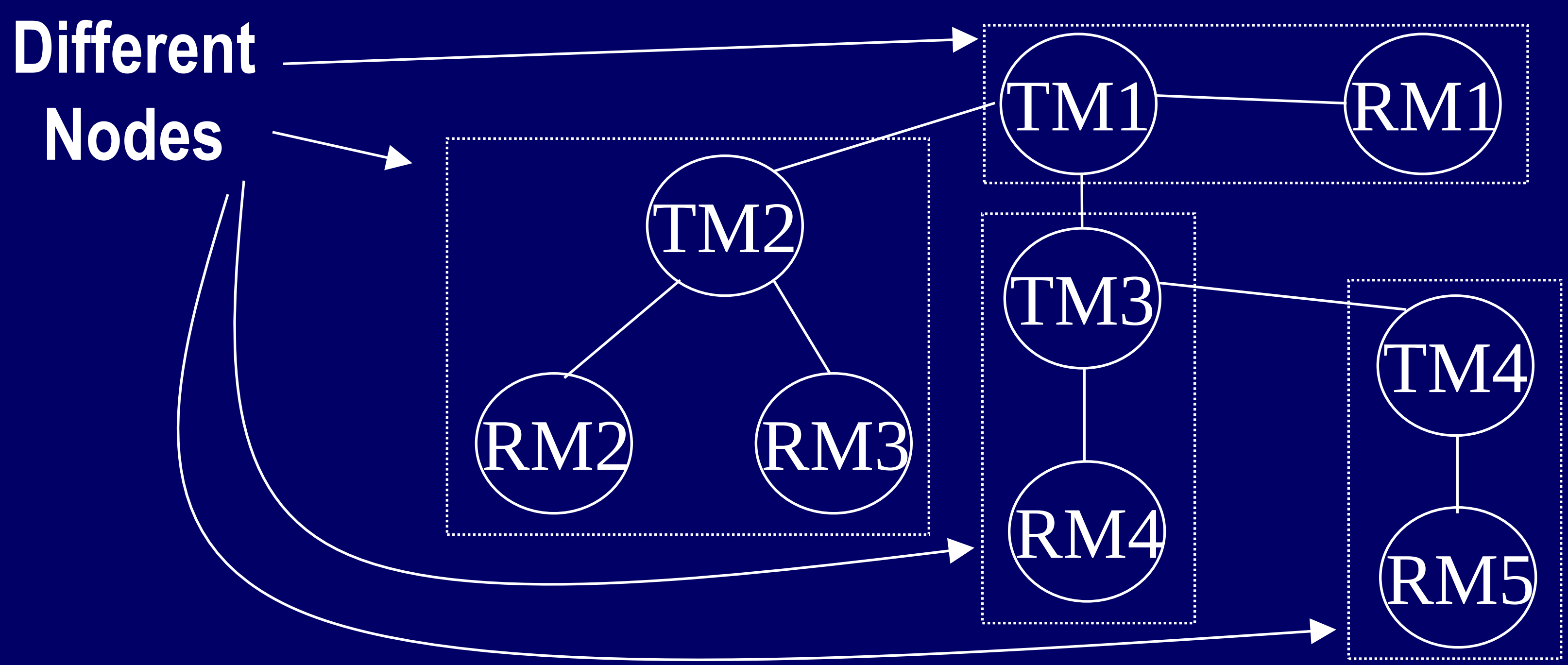
# Enlisting in a Transaction (cont'd)

- When an application A in a transaction T first calls an application B at another node, B must tell its local TM that the transaction has arrived.



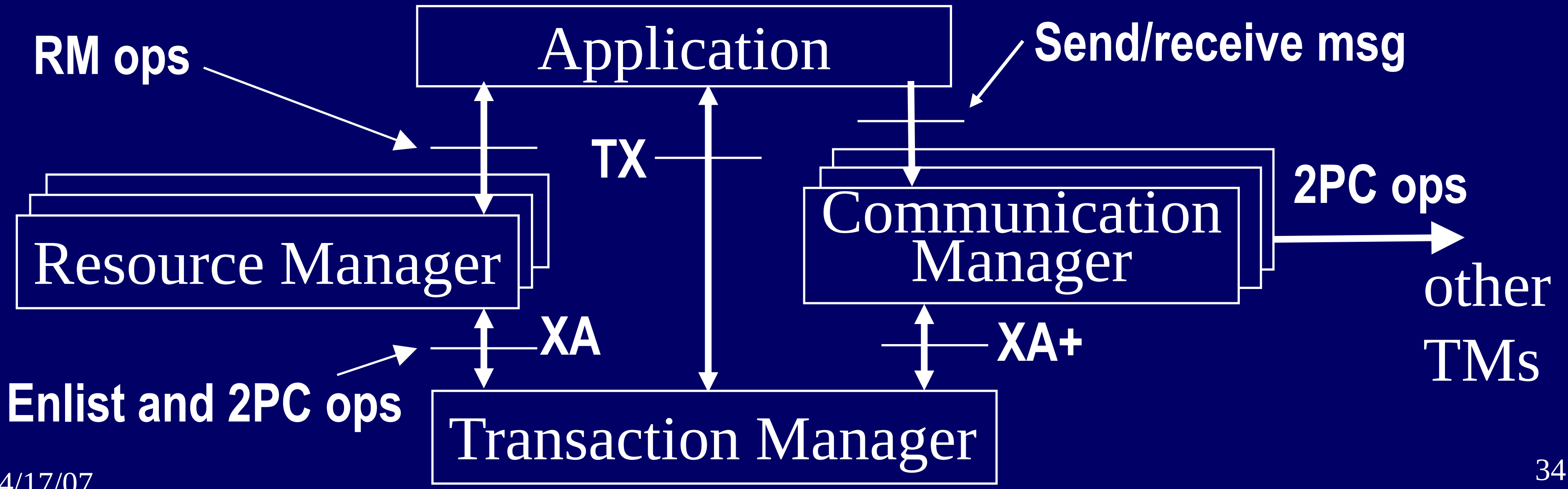
# Tree of Processes

- Application calls to RMs and other applications induces a <u>tree of processes</u>
- Each internal node is the coordinator for its descendants, and a participant to its parents.
- This adds delay to two-phase commit
- Optimization: flatten the tree, e.g. during phase 1

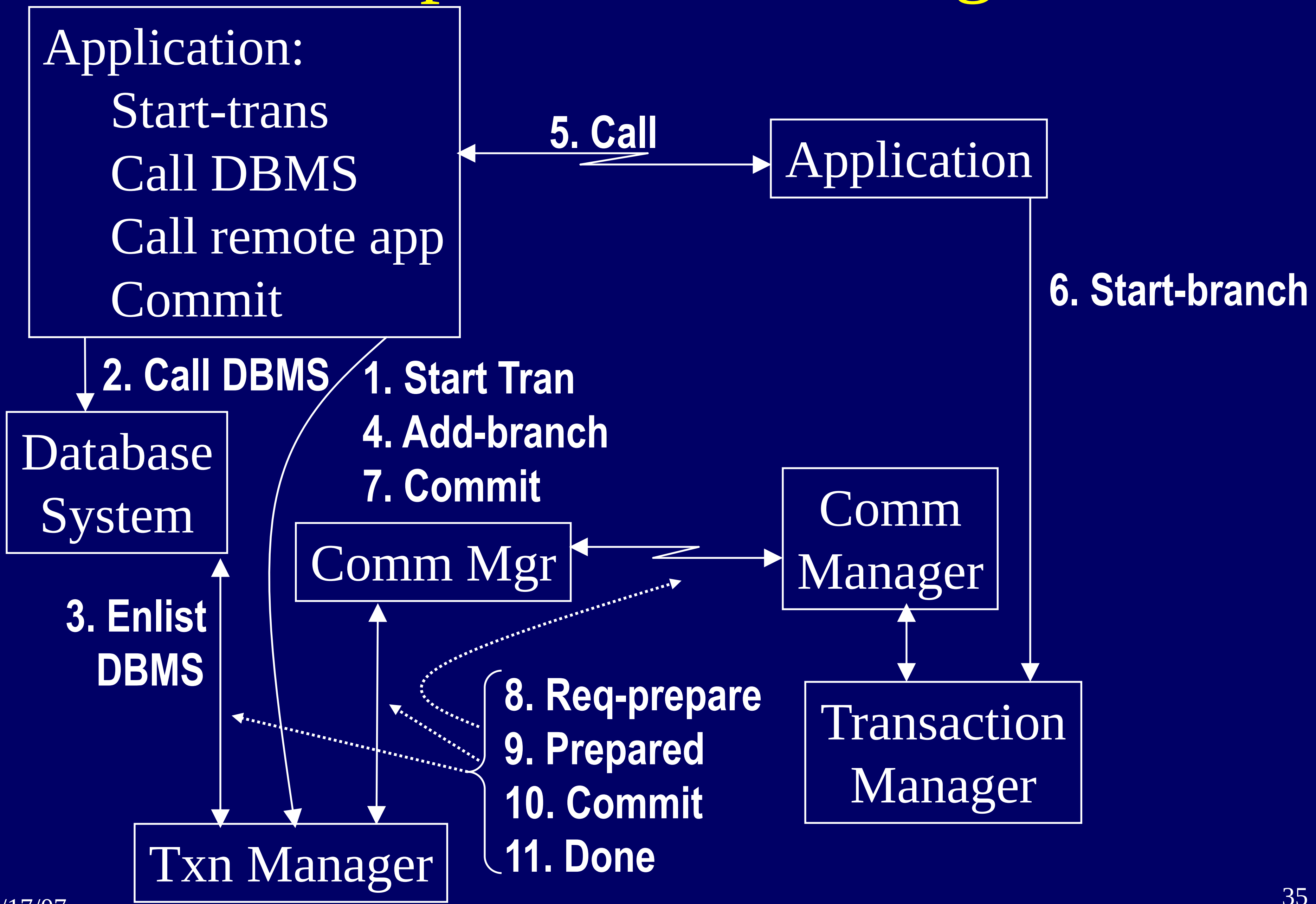


# Handling Multiple Protocols

- Communication managers solve the problem of handling multiple 2PC protocols by providing
  - a model for communication between address spaces
  - a wire protocol for two-phase commit
- But, expect restrictions on multi-protocol interoperation.
- The RM only talks to the TM-RM interface. The multi-protocol problem is solved by the TM vendor.



# Complete Walkthrough



# Customer Checklist

- Does your DBMS support 2PC?
- Does your execution environment support it? If so,
  - with what DBMSs?
  - Using what protocol(s)?
  - Do these protocols meet your interoperation needs?
- Is the TM-DBMS interface open (for home-grown DBMSs)?
- Can an operator commit/abort a blocked txn?
  - If so, is there automated support for reconciling mistakes?
  - Is there automated heuristic commit?

# 7.6 Three Phase Commit- The Idea

- 3PC prevents blocking in the absence of communications failures (unrealistic, but …). It can be made resilient to communications failures, but then it may block
- 3PC is <u>much</u> more complex than 2PC, but only marginally improves reliability — prevents some blocking situations.
- 3PC therefore is not used much in practice
- Main idea: becoming certain and deciding to commit are separate steps.
- 3PC ensures that if any operational process is uncertain, then <u>no</u> (failed or operational) process has committed.
- So, in the termination protocol, if the operational processes are all uncertain, they can decide to abort (avoids blocking).

# Three Phase Commit- The Protocol

1. (Begin phase 1) Coordinator C sends **Request-to-prepare** to all participants
2. Participants vote **Prepared** or **No**, just like 2PC.
3. If C receives **Prepared** from <u>all</u> participants, then (begin phase 2) it sends **Pre-Commit** to all participants.
4. Participants wait for **Abort** or **Pre-Commit**. Participant acknowledges **Pre-commit**.
5. After C receives acks from all participants, or times out on some of them, it (begin third phase) sends **Commit** to all participants (that are up)

# 3PC Failure Handling

- If coordinator times out before receiving **Prepared** from all participants, it decides to abort.
- Coordinator ignores participants that don't ack its **Pre-Commit.**
- Participants that voted **Prepared** and timed out waiting for **Pre-Commit** or **Commit** use the termination protocol.
- The termination protocol is where the complexity lies. (E.g. see [Bernstein, Hadzilacos, Goodman 87], Section 7.4)