# CONCURRENCY CONTROL

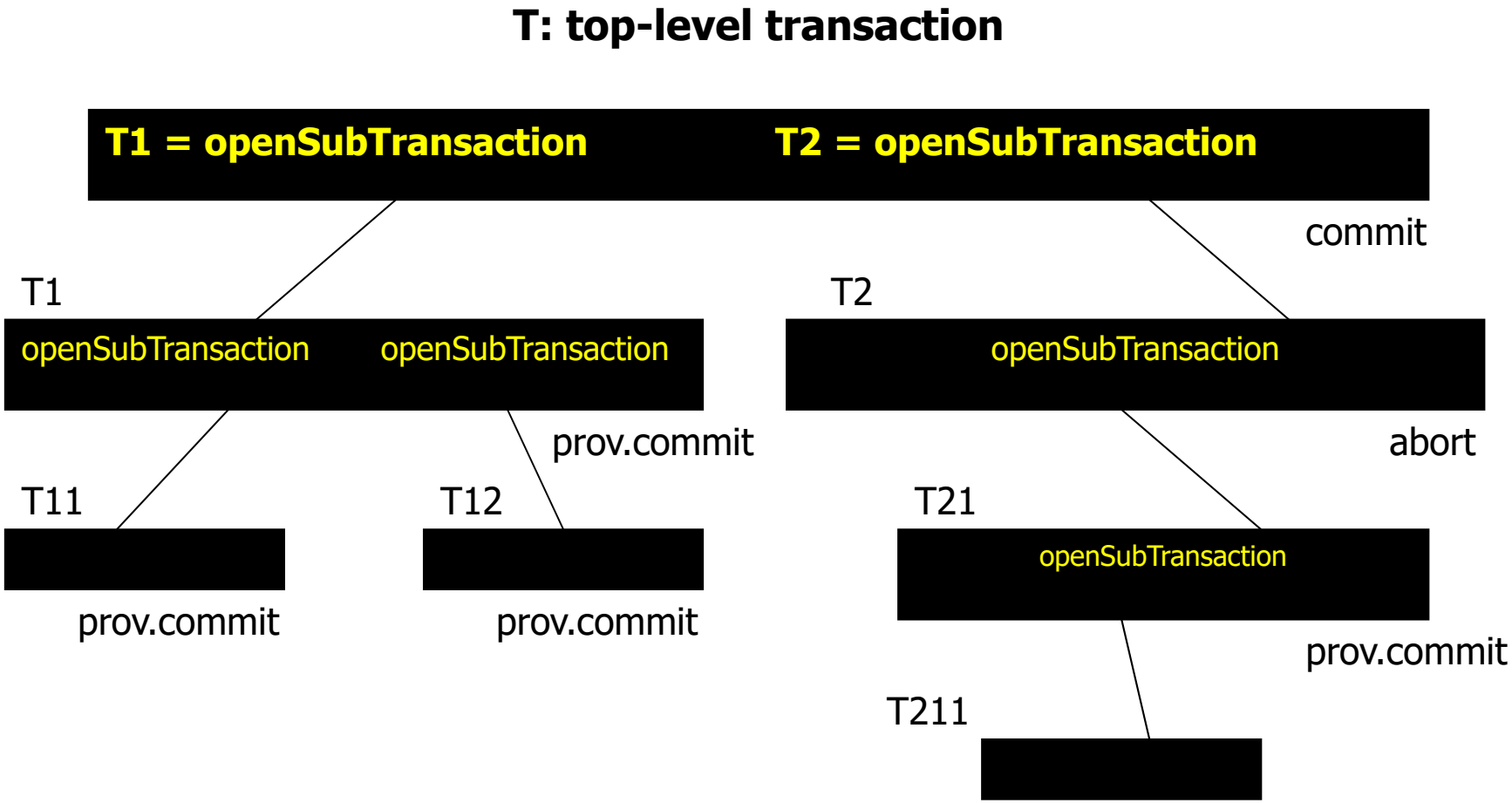
# CONTENT

- Methods for concurrency control
  - Locks
  - Optimistic concurrency control
  - Timestamp ordering
- Distributed Transactions
- Distributed Deadlock

# Transactions (ACID)

- **Atomic**: All or nothing. No intermediate states are visible.
- **Consistent**: system invariants preserved.
- **Isolated**: Two transactions do not interfere with each other. They appear as serial executions.
- **Durable**: The commit causes a permanent change.

# Nested transaction



# NESTED TRANSACTION

- The outermost transaction in a set of nested transactions is called the top-level transaction.
- Transactions other than the top-level transactions are called sub-transaction.
- Any sub-transaction appears atomic to its parent with respect to failures.
- Sub-transaction at the same level can run concurrently but their access to common objects is serialized.

# NESTED TRANSACTION

- Each sub-transaction can fail independently of its parent and of the other sub-transaction.
- When a sub-transaction aborts, the parent transaction can sometimes choose an alternative sub-transaction to complete its task.
- If all the tasks is done on same level, then it is called flat transaction.

# TRANSACTION & NESTED TRANSACTION

- Advantages of nested transaction:
  - Sub-transaction at one level (and descendent) may run concurrently with other sub-transaction: Additional concurrency in a transaction. If sub-transactions run in different servers, they can work parallel.
  - Subtransactions can commit or abort independently

# Schemes for Concurrency control

- Locking
  - Server attempts to gain an exclusive 'lock' that is about to be used by one of its operations in a transaction.
  - Can use different lock types (read/write for example)
  - Two-phase locking
- Optimistic concurrency control
- Time-stamp based concurrency control

# METHOD FOR CONCURRENCY CONTROL

- Lock:
  - Server attempts to lock any object that is about to use by client's transaction.
  - Requests to lock objects are suspended and wait until the objects are unlocked.
  - Serial equivalence: transaction is not allowed any new locks after it has release a lock.
    - Two-phase lock: growing phase (new locks are acquired), shrinking phase (locks are released).
  - Strict execution: locks are held until transaction commits/aborts (Strict two-phase locking).
  - Recoverability: locks must be held until all the objects it updated have been written to permanent storage.

# METHOD FOR CONCURRENCY CONTROL

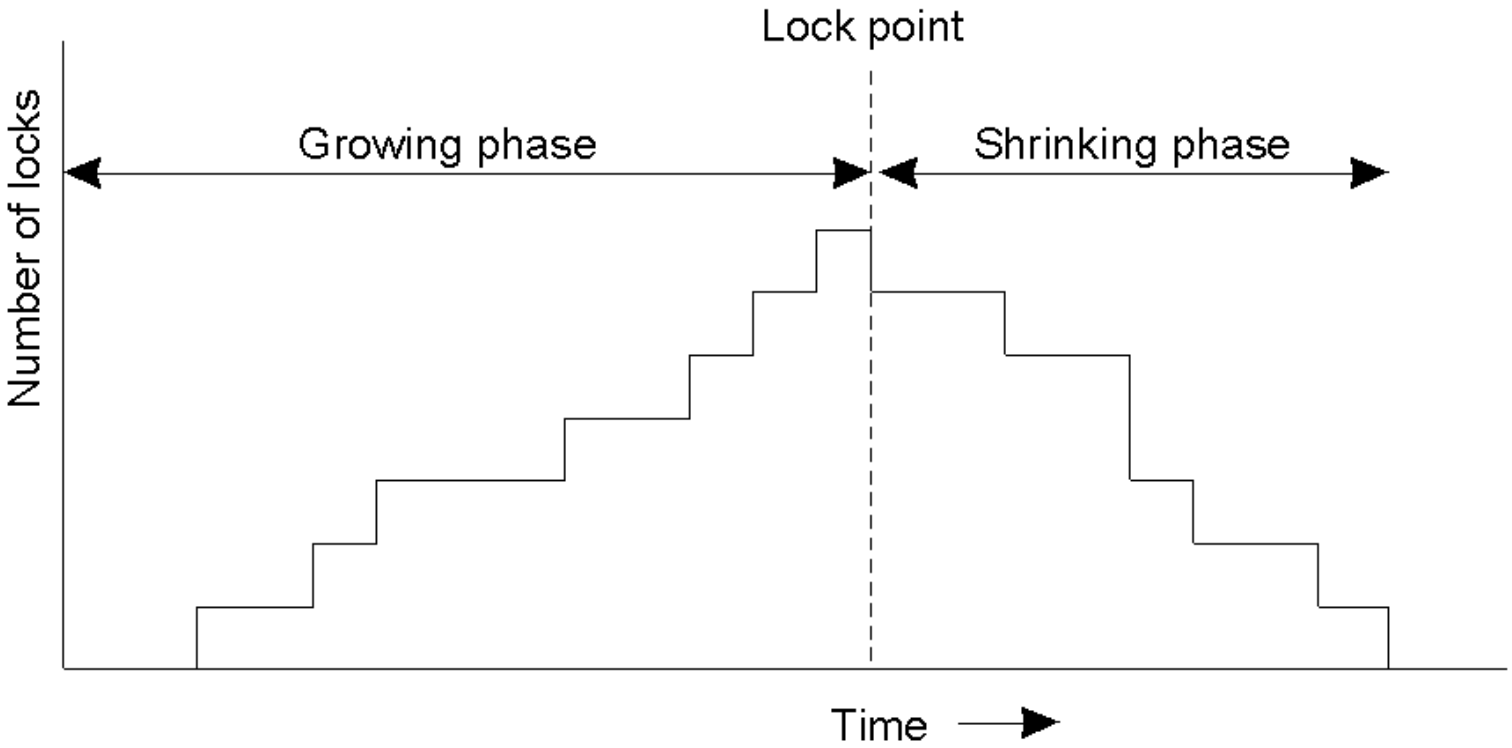
- Lock:
  - Simple exclusive lock reduces concurrency→ locking scheme for multiple transaction reading an object, single transaction writing an object.
  - Two types of locks used: read locks (shared lock) & write locks.
  - Operation conflict rules:
    - Request for a write lock is delayed by the presence of a read lock belonging to another transaction.
    - Request for either a read/write lock is delayed by the presence of a write lock belonging to another transaction.

# Transaction T and U with exclusive locks.

| TRANSACTION T | TRANSACTION U |
|---|---|
| balance = b.getBalance( );<br>b.setBalance(balance*1.1);<br>a.withdraw(balance/10); | balance = b.getBalance( );<br>b.setBalance(balance*1.1);<br>c.withdraw(balance/10); |
| openTransaction<br>balance = b.getBalance( ); lock B<br>b.setBalance(balance*1.1);<br>a.withdraw(balance/10); lock A<br><br>closeTransaction unlock A,B | <br><br><br>openTransaction<br>balance = b.getBalance( ); wait for T'lock on B<br>…<br>b.setBalance(balance*1.1); lock B<br>c.withdraw(balance/10) ; lock C<br>closeTransaction unlock B,C |

- Assumption: balance of ABC are not yet locked when transaction T and U starts.
- When T starts using B, then it is locked for B. subsequently when U starts using B, it is still locked for T and so U waits. When T committed, B is unlocked and C resumes : effective serialization

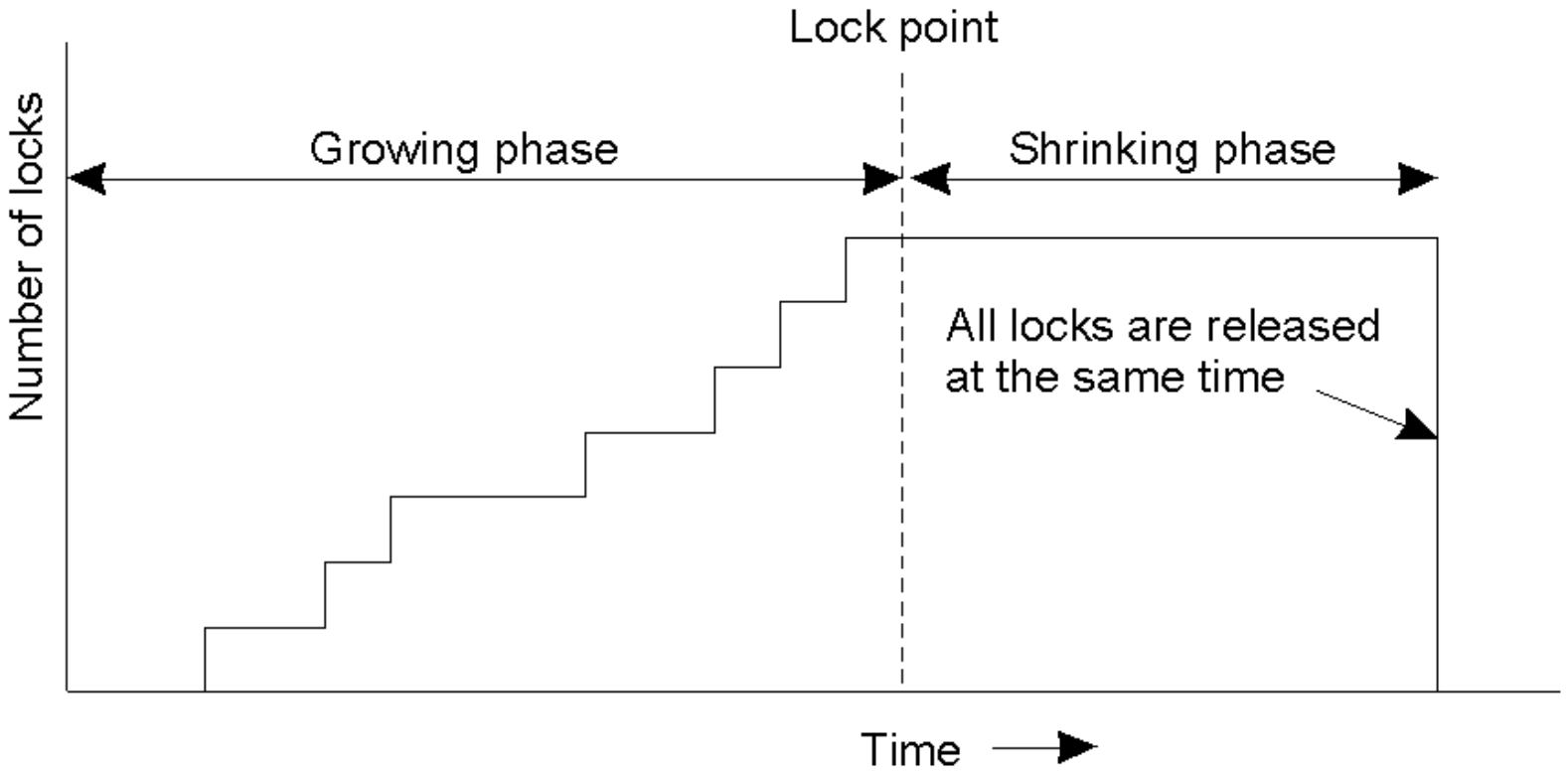
# Two-Phase Locking (1)




In two-phase locking, a transaction is not allowed to acquire any new locks after it has released a lock

# Strict Two-Phase Locking (2)

- Strict two-phase locking.



# Use of locks in strict two-phase locking

1. When an operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)
2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

# Lock compatibility

| For one object | | Lock requested | |
|---|---|---|---|
| | | Read | Write |
| Lock already set | none | OK | OK |
| | read | OK | wait |
| | write | Wait | wait |

# METHOD FOR CONCURRENT CONTROL

- Locking rule for nested transactions:
  - Locks that are acquired by a successful subtransaction is inherited by its parent & ancestors when it completes. Locks held until top-level transaction commits/aborts.
  - Parent transactions are not allowed to run concurrently with their child transactions.
  - Subtransactions at the same level are allowed to run concurrently.

# Method for concurrent control

- Deadlock:
  - Definition: A state in which each member of a group of transactions is waiting for some other member to release a lock.
  - Prevention:
    - Lock all the objects used by a transaction when it starts → not a good way.
    - Request locks on objects in a predefined order → premature locking & reduction in concurrency.

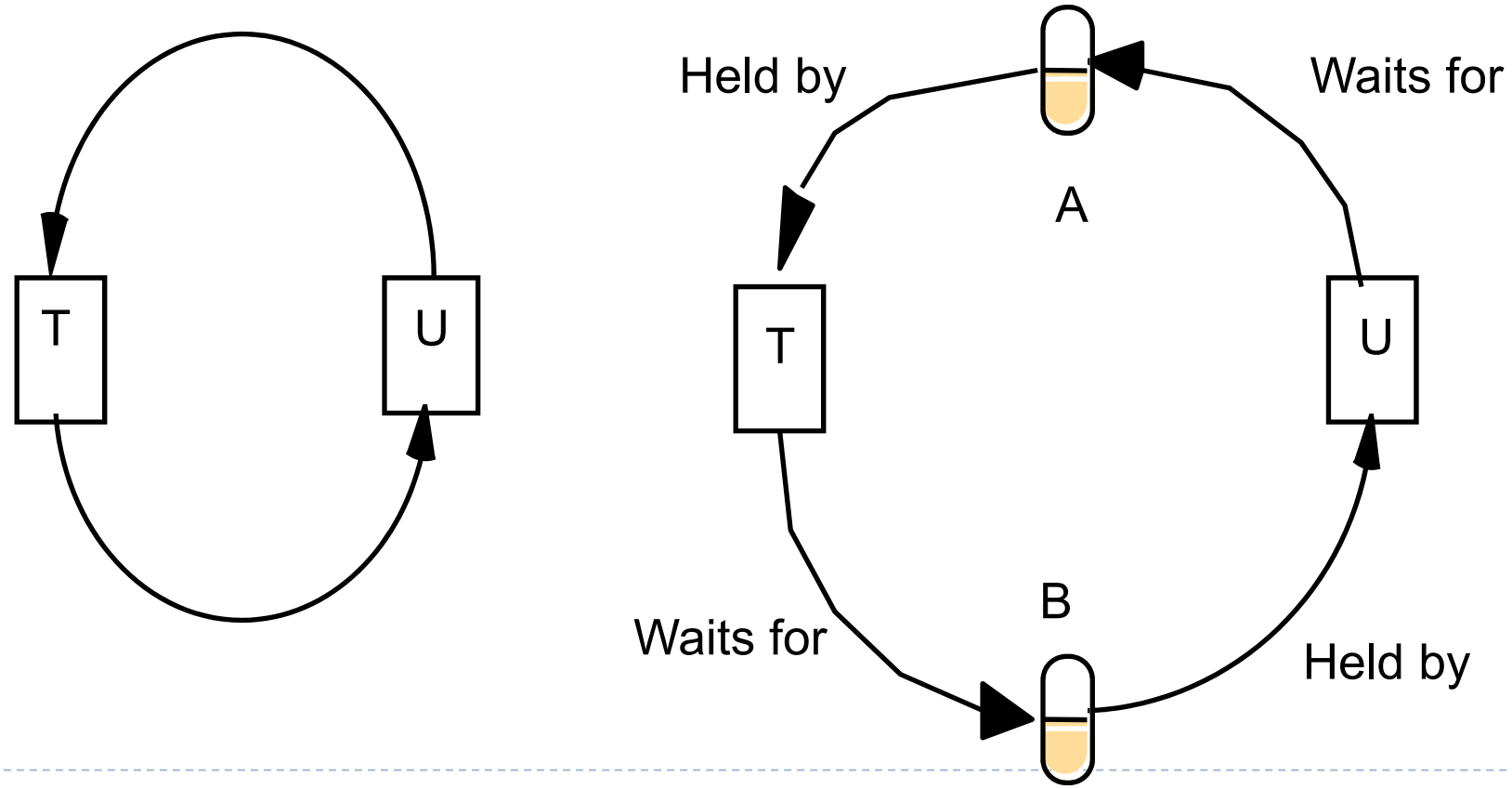
# METHOD FOR CONCURRENT CONTROL

- Deadlock:
  - Detection: Finding cycle in a wait-for graph→ select a transaction for aborting to break the cycle.
    - Choice of transaction to be aborted is not simple.
  - Timeouts: each lock is given a limited period in which it is invulnerable.
    - Transaction is sometimes aborted but actually there is no deadlock.
- If we use locking to implement concurrency control in transactions, we can get deadlocks (even within a single server)
- So we need to discuss:
  - Deadlock detection within a single system
  - Distributed deadlock

# Deadlock detection

- A deadlock occurs when there is a cycle in the *wait-for* graph of transactions for locks
- There may be more than one
- Resolve the deadlock by aborting one of the transactions ….
- E.g. the youngest, or the one involved in more than one cycle, or can even use "priority" ….

A cycle in a wait-for graph



# METHOD FOR CONCURRENCY CONTROL

- Drawbacks of locking:
  - Lock maintenance represents an overhead that is not present in systems that do not support concurrent access to shared data.
  - Deadlock. Deadlock prevention reduces concurrency. Deadlock detection or timeout not wholly satisfactory for use in interactive programs.
  - To avoid cascading aborts, locks cant be released until the end of the transaction. This may reduce significantly the potential for concurrency.

# METHOD FOR CONCURRENT CONTROL

- Optimistic concurrency control:
  - Is an alternative optimistic approach to the serialization of transactions that avoids the drawbacks of locking.
  - Idea: in most applications, the likelihood of two clients transactions accessing the same object is low.
  - Transactions are allowed to proceed as though there were no possibility of conflict with other transactions until the client completes its task and issues a close-Transaction request.

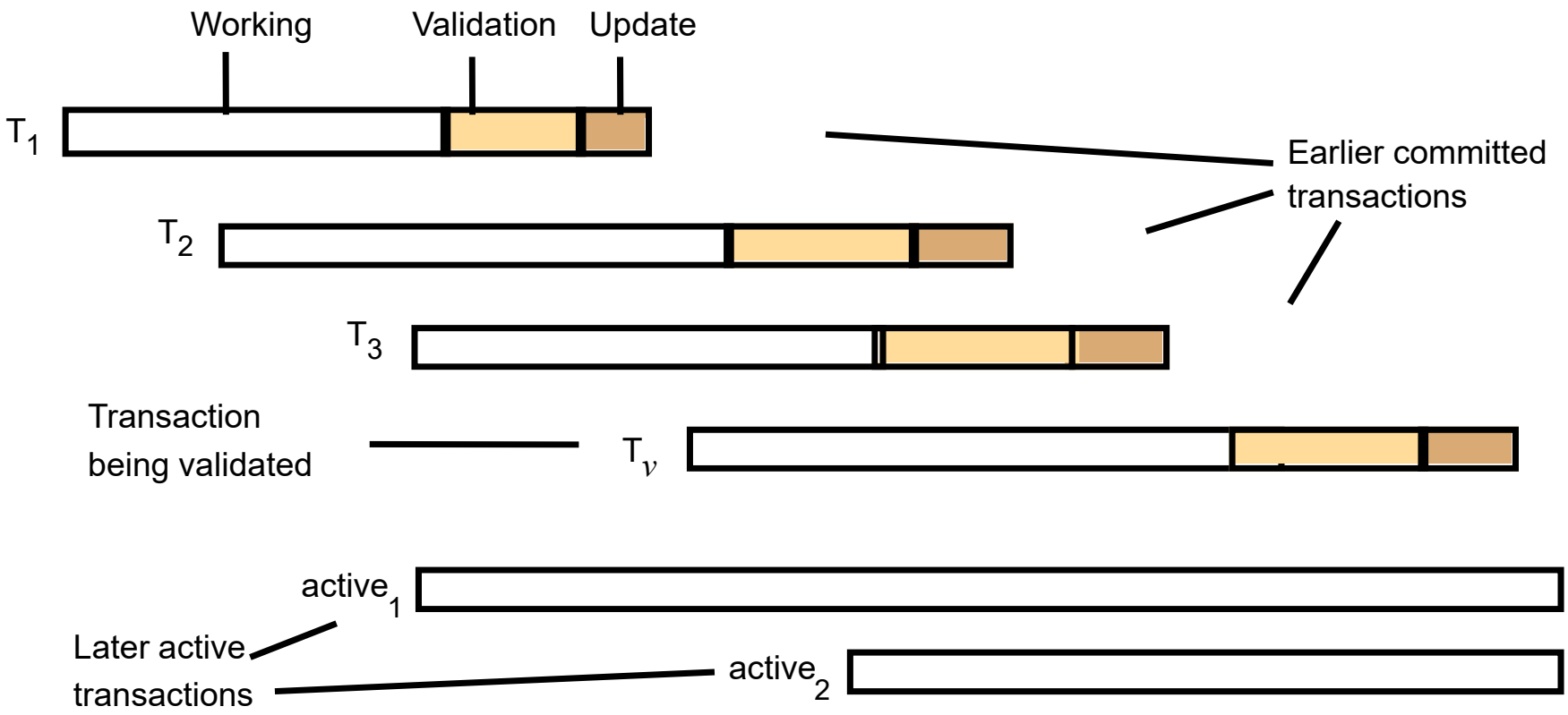
# METHOD FOR CONCURRENT CONTROL

- Optimistic concurrency control:
  - Each transaction has the following 3 phases:
    - **Working phase:** each transaction has a tentative version of each of the objects that it updates.
    - **Validation phase**: Once transaction is done, the transaction is validated to establish whether or not its operations on objects conflict with operations of other transactions on the same object. If not conflict, can commit; else some form of conflict resolution is needed and the transaction may abort.
    - **Update phase**: changes in tentative versions are made permanent if transaction is validated

# METHOD FOR CONCURRENT CONTROL

- Optimistic concurrency control:
  - **Validation of transactions**: use the read-write conflict rules to ensure that the scheduling of a transaction is serially equivalent with respect to all other overlapping transactions.
  - **Backward validation:** check the transaction undergoing validation with other preceding overlapping transactions (enter the validation phase before).
    - Read set of the transaction being validated is compared with the write sets of other transactions that have already committed.
  - **Forward validate**: check the transaction undergoing validation with other later transactions
    - Write set of the transaction being validated is compared with the read sets of other overlapping active transactions (still in working phase).

# Validation of transactions



# Schemes for Concurrency control

- Time-stamp based concurrency control
  - Each transaction is assigned a unique timestamp at the moment it starts
    - In distributed transactions, Lamport's timestamps can be used
  - Every data item has a timestamp
    - Read timestamp = timestamp of transaction that last read the item
    - Write timestamp = timestamp of transaction that most recently changed an item

# METHOD FOR CONCURRENT CONTROL

- Timestamp ordering:
  - Basic timestamp ordering rule:
    - A transaction's request to write an object is valid only if that object was last read and written by earlier transactions. A transaction's request to read an object is valid only if that object was last written by an earlier transactions
- Timestamp ordering write rule:

if ($T_c \geq$ maximum read timestamp on D &&

$T_c >$ write timestamp on committed version of D)

perform write operation on tentative version of D with write timestamp $T_c$

else */*write is too late*/*

abort transaction $T_c$

# METHOD FOR CONCURRENT CONTROL

- Timestamp ordering read rule:

If ($T_c$> write timestamp on committed version of D)

{ let $D_{selected}$ be the version of D with the maximum write timestamp ≤ Tc

if ($D_{selected}$ is committed)

perform read operation on the version $D_{selected}$

else

wait until the transaction that made version $D_{selected}$ commits or aborts then reapply the read rule

}

Else

abort transaction $T_c$

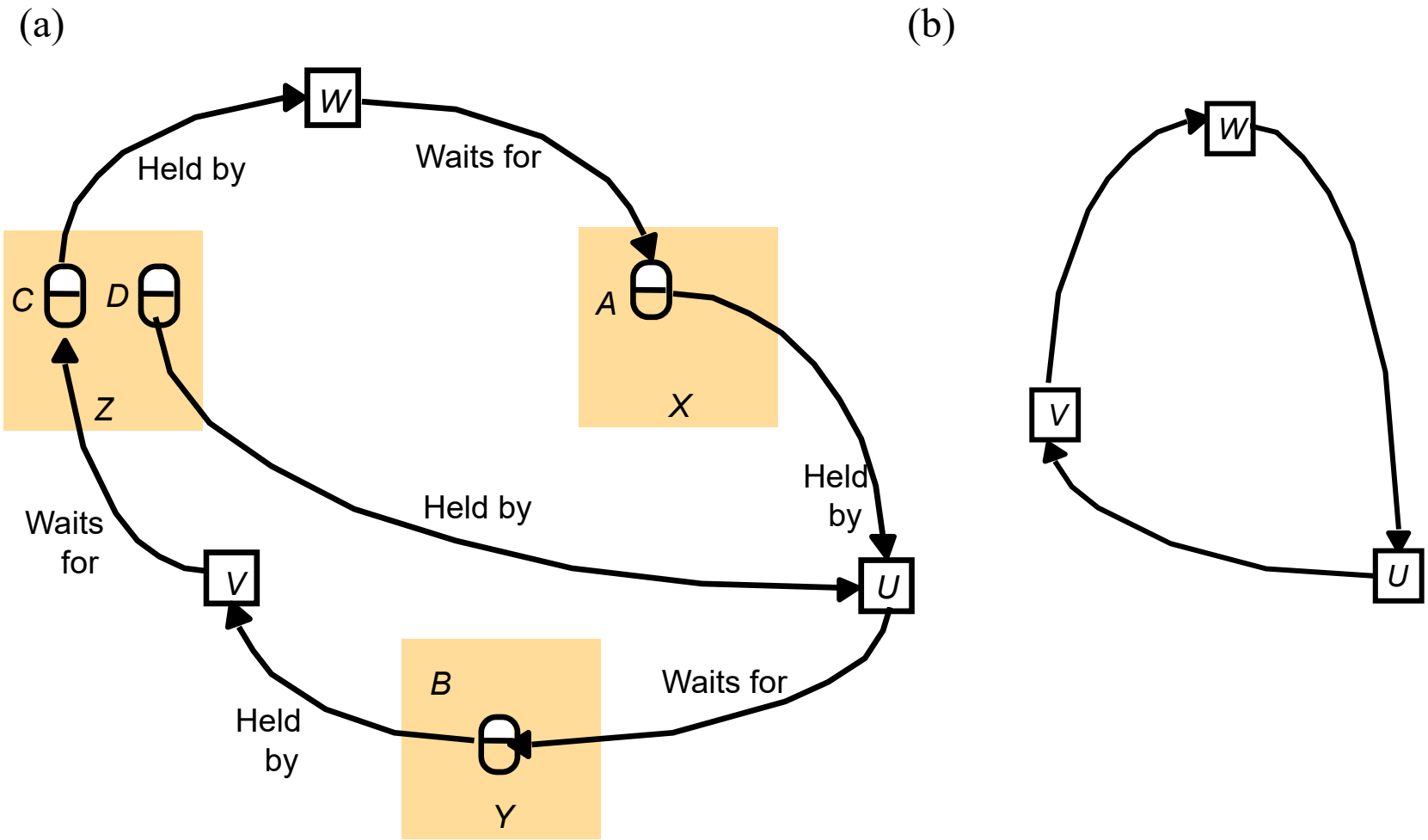
# Concurrency Control for Distributed Transactions

- Locking
  - Distributed deadlocks possible
- Timestamp ordering
  - Lamport time stamps
    - for efficiency it is required that timestamps issued by coordinators be roughly synchronized

# Distributed Deadlock

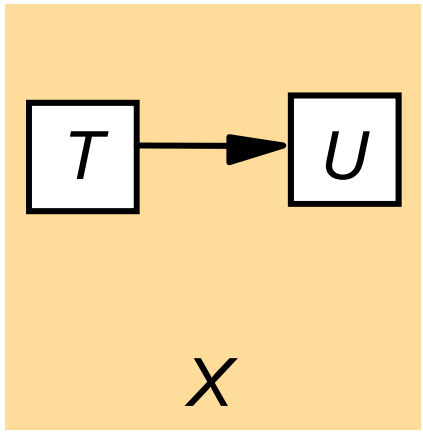
- Within a single server, allocating and releasing locks can be done so as to maintain a wait-for graph which can be periodically checked.
- With distributed transactions locks are held in different servers – and the loop in the entire wait-for graph will not be apparent to any one server
- One solution is to have a coordinator to which each server forwards its wait-for graph
- But centralised coordination is not ideal in a distributed system

# Distributed deadlock

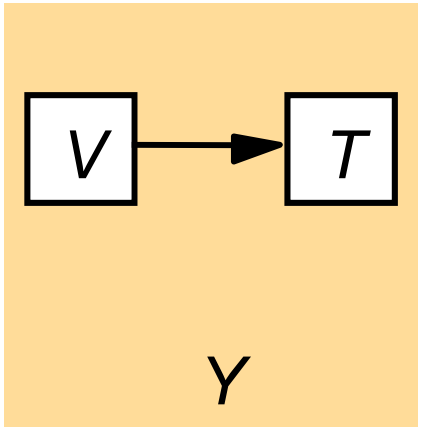


# Local and global wait-for graphs

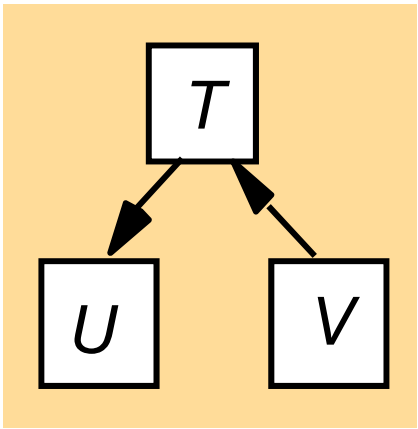
local wait-for graph




local wait-for graph




global deadlock detector



# Atomic Commit Protocols

- The atomicity of a transaction requires that when a distributed transaction comes to an end, either all of its operations are carried out or none of them
- Two phase commit (2PC)
- Three Phase Commit (3PC)
- Recovery….

# The two-phase commit protocol - 1

*Phase 1 (voting phase):*

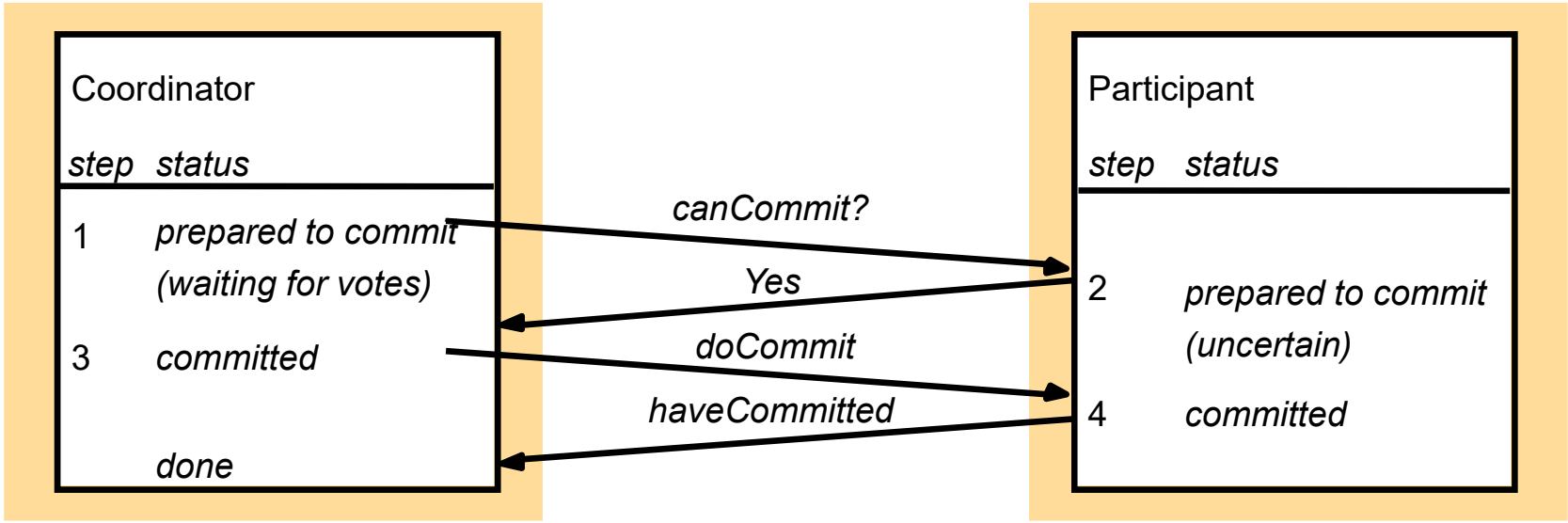
1. The coordinator sends a *canCommit? (VOTE_REQUEST)* request to each of the participants in the transaction.
2. When a participant receives a *canCommit*? request it replies with its vote *Yes (VOTE_COMMIT)* or *No* (*VOTE_ABORT*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No* the participant aborts immediately.

# The two-phase commit protocol - 2

*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 (GLOBAL_COMMIT)* request to each of the participants.
   (b) Otherwise the coordinator decides to abort the transaction and sends *doAbort (GLOBAL_ABORT)* 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.

# Communication in two-phase commit protocol



# Operations for two-phase commit protocol

*canCommit?(trans)-> Yes / No*

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.

*doAbort(trans)*

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

*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.

# Two-Phase Commit protocol - 3

**actions by coordinator:**

```
while START _2PC to local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected {
    wait for any incoming vote;
    if timeout {
        write GLOBAL_ABORT to local log;
        multicast  GLOBAL_ABORT to all participants;
        exit;
    }
    record vote;
}
if all participants sent VOTE_COMMIT and coordinator votes COMMIT{
    write GLOBAL_COMMIT to local log;
    multicast GLOBAL_COMMIT to all participants;
} else {
    write GLOBAL_ABORT  to local log;
    multicast GLOBAL_ABORT to all participants;
}
```

Outline of the steps taken by the coordinator in a two phase commit protocol

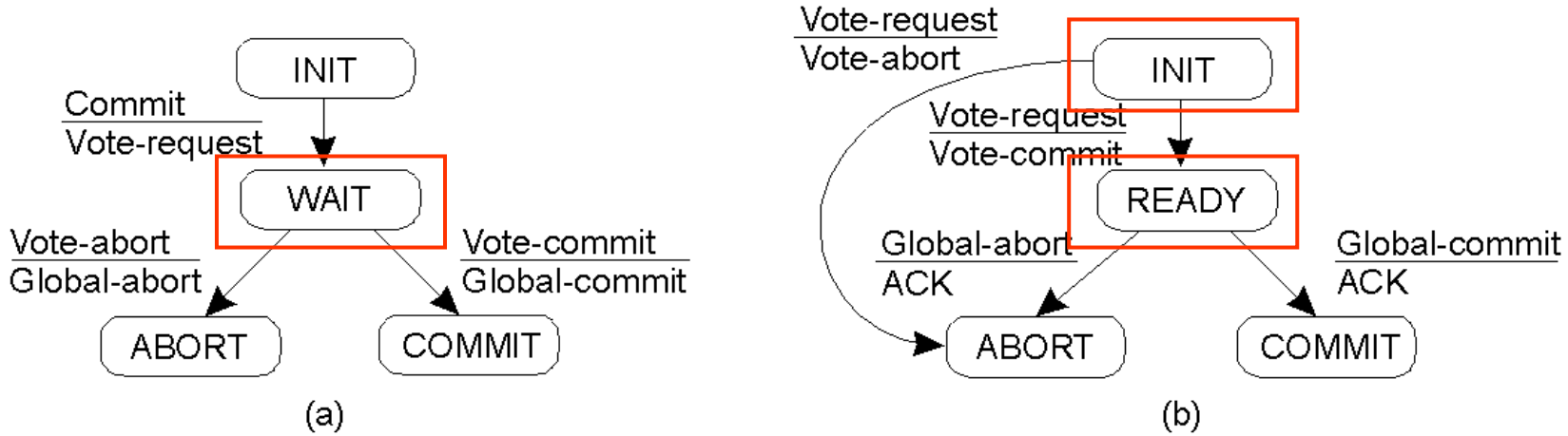
# Two-Phase Commit protocol - 4

Steps taken by participant process in 2PC.

**actions by participant:**

```
write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
    write VOTE_ABORT to local log;
    exit;
}
if participant votes COMMIT {
    write VOTE_COMMIT to local log;
    send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
        multicast DECISION_REQUEST to other participants;
        wait until DECISION is received; /* remain blocked */
        write DECISION to local log;
    }
    if DECISION == GLOBAL_COMMIT
        write GLOBAL_COMMIT to local log;
    else if DECISION == GLOBAL_ABORT
        write GLOBAL_ABORT to local log;
} else {
    write VOTE_ABORT to local log;
    send  VOTE ABORT to coordinator;
}
```

# Two-Phase Commit protocol - 5




a) The finite state machine for the coordinator in 2PC.

b) The finite state machine for a participant.

If a failure occurs during a 'blocking' state (red boxes), there needs to be a recovery mechanism.

# Two Phase Commit Protocol - 6

Recovery

- 'Wait' in Coordinator – use a time-out mechanism to detect participant crashes. Send GLOBAL_ABORT
- 'Init' in Participant – Can also use a time-out and send VOTE_ABORT
- 'Ready' in Participant P – abort is not an option (since already voted to COMMIT and so coordinator might eventually send GLOBAL_COMMIT). Can contact another participant Q and choose an action based on its state.

| State of Q | Action by P |
|---|---|
| COMMIT | Transition to COMMIT |
| ABORT | Transition to ABORT |
| INIT | Both P and Q transition to ABORT<br>(Q sends VOTE_ABORT) |
| READY | Contact more participants. If all participants are 'READY', must wait for coordinator to recover |

# Two-Phase Commit protocol - 7

**actions for handling decision requests:** /* executed by separate thread */

```
while true {
    wait until any incoming DECISION_REQUEST is received; /* remain blocked */
    read most recently recorded STATE from the local log;
    if STATE == GLOBAL_COMMIT
        send GLOBAL_COMMIT to requesting participant;
    else if STATE == INIT or STATE == GLOBAL_ABORT
        send GLOBAL_ABORT to requesting participant;
    else
        skip;  /* participant remains blocked */
```

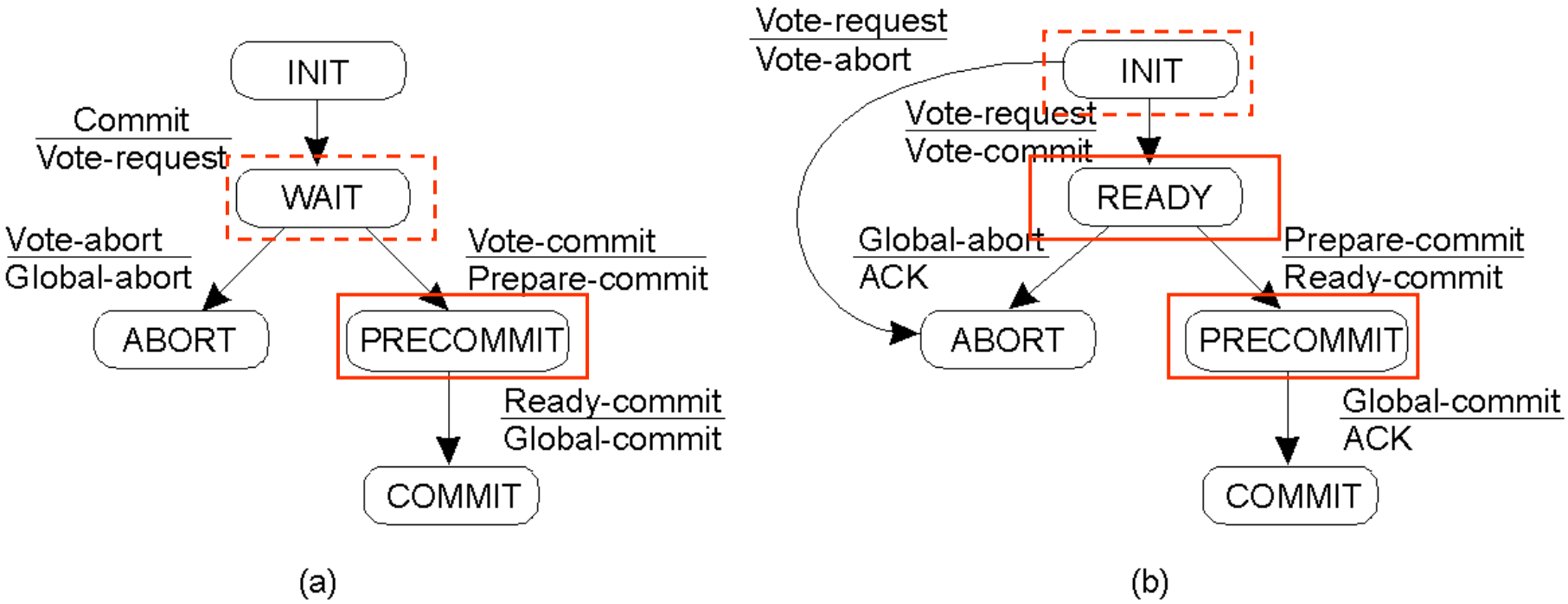
Steps taken for handling incoming decision requests.

# Three Phase Commit protocol - 1

- Problem with 2PC
  - If coordinator crashes, participants cannot reach a decision, stay blocked until coordinator recovers

- Three Phase Commit3PC
  - There is no single state from which it is possible to make a transition directly to either COMMIT or ABORT states
  - There is no state in which it is not possible to make a final decision, and from which a transition to COMMIT can be made

# Three-Phase Commit protocol - 2




a) Finite state machine for the coordinator in 3PC

b) Finite state machine for a participant

# Three Phase Commit Protocol - 3

- 'Wait' in Coordinator – same
- 'Init' in Participant – same
- 'PreCommit' in Coordinator – Some participant has crashed but we know it wanted to commit. GLOBAL_COMMIT the application knowing that once the participant recovers, it will commit.
- 'Ready' or 'PreCommit' in Participant P – (i.e. P has voted to COMMIT)

| State of Q | Action by P |
|---|---|
| PRECOMMIT | Transition to PRECOMMIT. If all participants in PRECOMMIT, can COMMIT the transaction |
| ABORT | Transition to ABORT |
| INIT | Both P (in READY) and Q transition to ABORT (Q sends VOTE_ABORT) |
| READY | Contact more participants. If can contact a majority and they are in 'Ready', then ABORT the transaction.<br>If the participants contacted in 'PreCommit' it is safe to COMMIT the transaction |

*Note: if any participant is in state PRECOMMIT, it is impossible for any other participant to be in any state other than READY or PRECOMMIT.*