15-440 Distributed Systems

Lecture 09 – Distributed Concurrency Management

The Two Phase Commit Protocol

Thursday, September 27th, 2018

Logistics Updates

- P1 Part A checkpoint (was due 9/25)
  - Part A due: Saturday 10/6 (6-week drop deadline 10/8)
  - Part B due: Tuesday 10/16
- HW2 will be released 10/01
  - HW 2 due: Friday, 10/10 (tentative, maybe 10/12)
  - (*No Late Days*) => time to prepare for Mid term
- We're currently grading HW1
  - HW1 solutions should come online soon

Today's Lecture Outline

- Transactions and Consistency
  - Database terminology
- Part I: Single Server Case
  - (not covered well in book)
  - Two Phase Locking
- Part II: Distributed Transactions
  - Two Phase Commit (Tanenbaum 8.5)

Assumptions for Today

a) Ignore failures in first half
   - Concurrency is our main concern

b) To deal with failures
   - Assume a form of logging, where every machine writes information down *before* operating on it, to recover from simple failures. Recover after failure.
   - Next lecture: Logging and Crash Recovery

Database transactions

- Background: Database Researchers
- Defined: “Transactions”
  - Collections of Reads + Writes to Global State
  - Appear as a single, “indivisible” operation
  - Standard Models for Reliable Storage (visit later)
- Desirable Characteristics of Transactions
  - Atomicity, Consistency, Isolation, Durability
  - Also referred to as the “ACID” Acronym!

Transactions: ACID Properties

- **Atomicity**: Each transaction completes in its entirely, or is aborted. If aborted, should not have have effect on the shared global state.
  - Example: Update account balance on multiple servers
- **Consistency**: Each transaction preserves a set of invariants about global state. (Nature of invariants is system dependent).
  - Example: in a bank system, law of conservation of $$

Transactions: ACID Properties

- **Isolation**: Also means serializability. Each transaction executes as if it were the only one with the ability to RD/WR shared global state.
- **Durability**: Once a transaction has been completed, or "committed" there is no going back. In other words there is no "undo".
- Transactions can also be nested
- "Atomic Operations" => Atomicity + Isolation

A Transaction Example: Bank

- Array `Bal[i]` stores balance of Account "`i`"
- **`Implement: xfer, withdraw, deposit`**

```
xfer(i, j, v):
    if withdraw(i, v):
        deposit(j, v)
    else
        abort
```

```
withdraw(i, v):
    b = Bal[i]          // Read
    if b >= v           // Test
        Bal[i] = b-v    // Write
        return true
    else
        return false
```

```
deposit(j, v):
    Bal[j] += v
```

A Transaction Example: Bank

```
xfer(i, j, v):
   if withdraw(i, v):
      deposit(j, v)
   else
      abort
```

```
withdraw(i, v):
   b = Bal[i]        // Read
   if b >= v         // Test
      Bal[i] = b-v   // Write
      return true
   else
      return false
```

```
deposit(j, v):
   Bal[j] += v
```

- Imagine: Bal[x] = 100, Bal[y]=Bal[z]=0
  - Two transactions => `T1:xfer(x,y,60), T2: xfer(x,z,70)`
  - ACID Properties: T1 or T2 in some serial order
    - T1; T2: T1 succeeds; T2 Fails. `Bal[x]=40, Bal[y]=60`
    - T2; T1: T2 succeeds; T1 Fails. `Bal[x]=30, Bal[z]=70`
  - What if we didn't take care? Is there a race condition?
    - Updating `Bal[x]` with Read/Write interleaving of T1,T2

A Transaction Example: Bank

```
xfer(i, j, v):
   if withdraw(i, v):
      deposit(j, v)
   else
      abort
```

```
withdraw(i, v):
  b = Bal[i]        // Read
  if b >= v         // Test
     Bal[i] = b-v   // Write
     return true
  else
     return false
```

```
sumbalance(i, j, k):
   return Bal[i]+Bal[j]+ Bal[k]
```

```
deposit(j, v):
   Bal[j] += v
```

- Imagine: Bal[x] = 100, Bal[y]=Bal[z]=0
  - Two transactions => `T1:xfer(x,y,60), T2: xfer(x,z,70)`
  - ACID violation: Not Isolated, Not Consistent
    - Updating `Bal[x]` with Read/Write interleaving of T1,T2
    - `Bal[x] = 30 or 40;  Bal[y] = 60; Bal [z] = 70`
  - For Consistency, implemented `sumbalance()`
    - State invariant sumbalance=100 violated! We created $$

Implement transactions with locks

- Use locks to wrap xfer

```
xfer(i, j, v):
   lock()
   if withdraw(i, v):
      deposit(j, v)
   else
      abort
   unlock()
```

However, is this the correct approach? (Hint: efficiency)

Sequential bottleneck due to global lock. Solution?

```
xfer(i, j, v):
   lock(i)
   if withdraw(i, v):
      unlock(i)
      lock(j)
      deposit(j, v)
      unlock(j)
   else
      unlock(i)
      abort
```

Is this fixed then?

No, consistency violation. sumbalance() after unlock(i)

Implement transactions with locks

Fix: Release locks when update of all state variables complete.

```
xfer(i, j, v):
    lock(i)
    if withdraw(i, v):
        lock(j)
        deposit(j, v)
        unlock(i);
        unlock(j)
    else
        unlock(i)
        abort
```

Are we done then?

Nope, deadlock.

Bal[x]=Bal[y]=100
xfer(x,y,40) and
xfer (y, x, 30)

Implement transactions with locks

Insight: Need unique global order for acquiring locks.

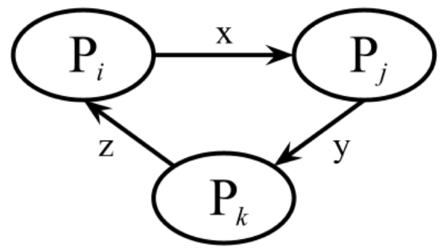
```
xfer(i, j, v):
   lock(min(i,j); lock(max (i,j))
   if withdraw(i, v):
      deposit(j, v)
      unlock(i); unlock(j)
   else
      unlock(i); unlock(j)
      abort
```

This **works**. :)

Motivation for 2-Phase Locking

Acquiring Locks in a Unique Order

- Consider “Wait-for” graph for state of locks
  - Vertices represent transactions
  - Edge from vertex i to vertex j if transaction i is waiting for lock held by transaction j.
- What does a cycle mean?



- Can a cycle occur if we acquire locks in unique order?
  - No. Label edges with its lock ID. For any cycle, there must be some pair of edges (i, j), (j, k) labeled with values x & y. As k holds y, but waits for x: y<x.
  - Transaction j is holding lock x and it wants lock y, so y > x.
  - Implies that j is not acquiring its lock in proper order.
- General scheme: 2-phase locking
  - More precisely: strong strict two phase locking

2-Phase Locking Variant

- General 2-phase locking
  - Phase 1: Acquire or Escalate Locks (e.g. read => write)
  - Phase 2: Release or de-escalate lock
- Strict 2-phase locking
  - Phase 1: (same as before)
  - Phase 2: Release WRITE lock at end of transaction only
- Strong Strict 2-phase locking
  - Phase 1: (same as before)
  - Phase 2: Release ALL locks at end of transaction only.
  - Most common version, required for ACID properties

2-Phase Locking

- Why not always use strong-strict 2-phase locking?
  - A transaction may not know the locks it needs in advance

```
if Bal(yuvraj) < 100:
   x = find_richest_prof()
   transfer_from(x, yuvraj)
```

- Other ways to handle deadlocks
  - Lock manager builds a “waits-for” graph. On finding a cycle, choose offending transaction and force abort
  - Use timeouts: Transactions should be short. If hit time limit, find transaction waiting for a lock and force abort.

Transactions – split into 2 phases

- Phase 1: Preparation:
  - Determine what has to be done, how it will change state, without actually altering it.
  - Generate Lock set "L"
  - Generate List of Updates "U"
- Phase 2: Commit or Abort
  - Everything OK, then update global state
  - Transaction cannot be completed, leave global state as is
  - In either case, RELEASE ALL LOCKS

Example

```
xfer(i, j, v):
    L={i,j} // Locks
    U=[]    //List of Updates
    begin(L) //Begin transaction, Acquire locks
    bi = Bal[i]
    bj = Bal[j]
    if bi >= v:
        Append(U,Bal[i] <- bi – v)
        Append(U, Bal[j] <- bj + v)
        commit(U,L)
    else
        abort(L)
```

Question: So, what would “commit” and ”abort” look like?

```
commit(U,L):
   Perform all updates in U
   Release all locks in L
```

```
abort(L):
   Release all locks in L
```

Today's Lecture Outline

- Consistency for multiple-objects, multiple-servers
- Part I: Single Server Case
  - (not covered well in book)
  - Two Phase Locking
- Part II: Distributed Transactions
  - Two Phase Commit (Tanenbaum 8.6)

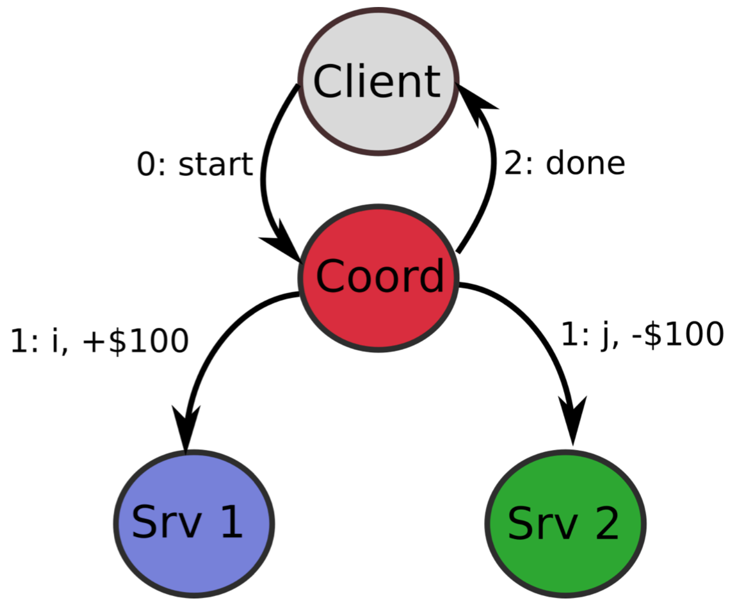
Distributed Transactions?

- Partition databases across multiple machines for scalability
  - (E.g., machine 1 responsible for account i,
machine 2 responsible for account j)
- Transaction often touch more than one partition
- How do we guarantee that all of the partitions commit the transactions or none commit the transactions?
  - Transferring money from i to j.
  - Requirement: **both** banks/machines do it, or **neither**

Enabling Distributed Transactions

- Similar idea as before, but:
  - State spread across servers (maybe even WAN)
  - Failures
- Overall Idea:
  - Client initiates transaction. Makes use of “coordinator”
  - All other relevant servers operate as “participants”
  - Coordinator assigns unique transaction ID (TID)
- Strawman solution
- 2-phase commit protocol

Strawman solution



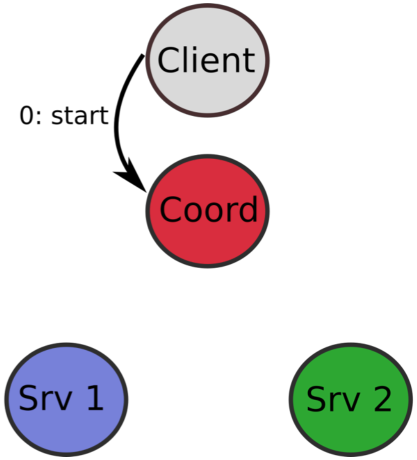
- Even without failures, a lot can go wrong
- Account j on Srv 2 has only $90
- Account j doesn't exist!
- Violates which part of ACID

2-Phase Commit

- Phase 1: Prepare & Vote
  - Participants figure out all state changes
  - Each determines if it can complete the transaction
  - Communicate with coordinator

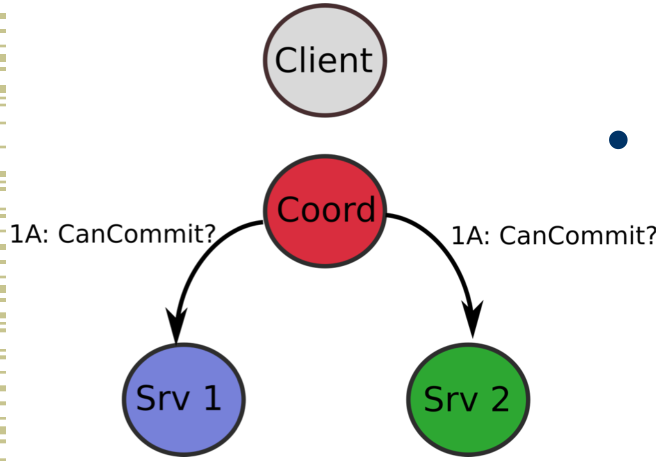
- Phase 2: Commit
  - Coordinator broadcasts to participants: COMMIT / ABORT
  - If COMMIT, participants make respective state changes

Implementing 2-Phase Commit



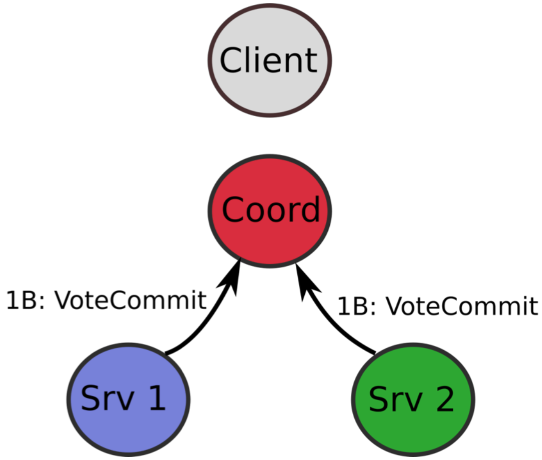
- Implemented as a set of messages

Implementing 2-Phase Commit



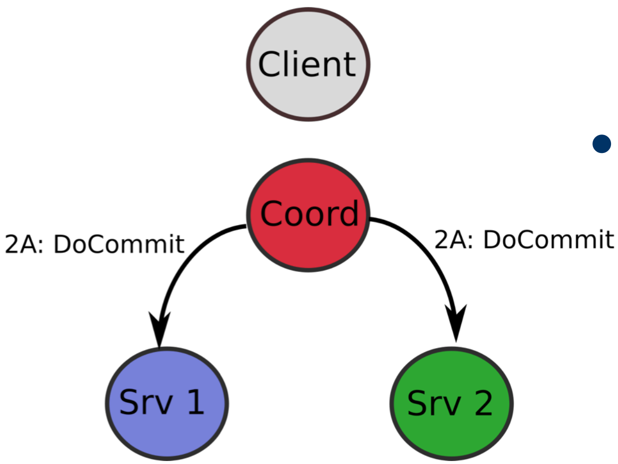
- Implemented as a set of messages
- Messages in first phase
  - A: Coordinator sends “CanCommit?” to participants

Implementing 2-Phase Commit



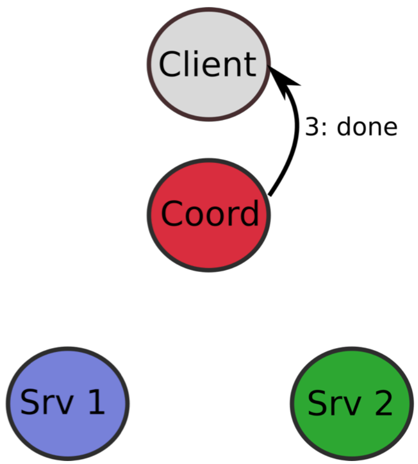
- Implemented as a set of messages
- Messages in first phase
  - A: Coordinator sends "CanCommit?" to participants
  - B: Participants respond: "VoteCommit" or "VoteAbort"

Implementing 2-Phase Commit



- Implemented as a set of messages
- Messages in first phase
  - A: Coordinator sends “CanCommit?” to participants
  - B: Participants respond: “VoteCommit” or “VoteAbort”
- Messages in the second phase
  - A: All “VoteCommit”: , Coord sends “DoCommit”
  - If any “VoteAbort”: abort transaction. Coordinator sends “DoAbort” to everyone => release locks

Implementing 2-Phase Commit



- Implemented as a set of messages
- Messages in first phase
  - A: Coordinator sends “CanCommit?” to participants
  - B: Participants respond: “VoteCommit” or “VoteAbort”
- Messages in the second phase
  - A: All “VotedCommit”: , Coord sends “DoCommit”
  - If any “VoteAbort”: abort transaction. Coordinator sends “DoAbort” to everyone => release locks

Example for 2PC

- Bank Account “i” at Server 1, “j” at Server 2.

```
L={i}
Begin(L) // Acq. Locks
U=[]  //List of Updates
b=Bal[i]
if b >= v:
   Append(U, Bal[i] <- b – v)
   vote commit
else
   vote abort
```

Server 1 implements transaction

```
L={j}
Begin(L) // Acq. Locks
U=[]  //List of Updates
b=Bal[j]
Append(U, Bal[j] <- b + v)
vote commit
```

Server 2 implements transaction

Server 2 can assume that the account of “i” has enough money, otherwise whole transaction will abort.

What about locking? Locks held by individual participants
- Acquire lock at start of prep process, release at Commit/Abort

Properties of 2-Phase Commit

- Correctness
  - Neither can commit unless both agreed to commit
- Performance
  - 3N messages per transaction
- How to handle failure?
  - Timeouts → performance bad in case of failure!

Deadlocks and Livelocks

- Distributed deadlock
  - Cyclic dependency of locks by transactions across servers
  - In 2PC this can happen if participants unable to respond to voting request (e.g. still waiting on a lock on its local resource)
  - Handled with a timeout. Participants times out, then votes to abort. Retry transaction again.
    - Addresses the deadlock concern
    - However, danger of LIVELOCK – keep trying!

Timeout and Failure Cases 1

- Coordinator times out after “CanCommit?”
  - Hasn’t sent any commit messages, safely abort
  - Conservative. Why?
  - Preserve correctness, sacrifice performance

- Participant times out after “VoteAbort”
  - Can safely abort unilaterally.
  - Why?

Timeout and Failure Cases 2

- Participant times out after “VoteCommit”
  - Are unilateral decisions possible? Commit, Abort?
  - Participant could wait forever

- Solution: ask another participant (gossip protocol)
  - Learn coordinator’s decision: do the same
    - Assumption: non-Byzantine failure model
  - Other participant hasn’t voted: abort is safe. Why?
    - Coordinator has not made decision
  - No reply or other participant also “VoteCommit”: wait
    - 2PC is “blocking protocol” → 3PC in book.

2 Phase Commit in Practice

2PC widely used in practice

NDB Cluster

Logging and Crash Recovery

- Crucial to handle crashes / reboots
  (next lecture)
- Very powerful and resilient when paired with RAID
  (3 lectures from now)

Summary

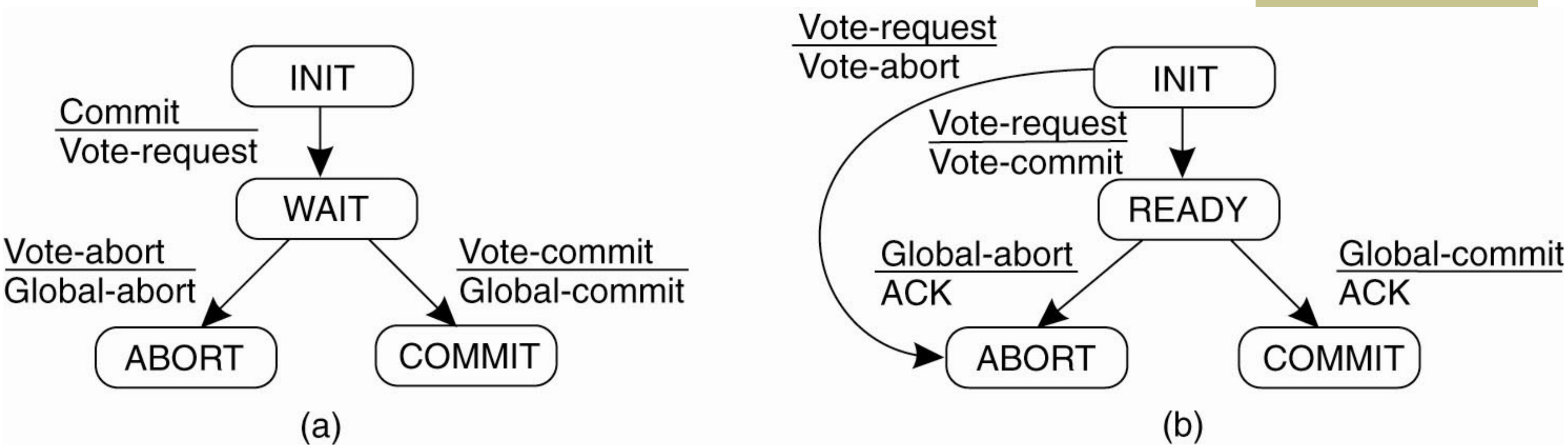
- Distributed consistency management
- ACID Properties desirable
- Single Server case: use locks + 2-phase locking (strict 2PL, strong strict 2PL), transactional support for locks
- Multiple server distributed case: use 2-phase commit for distributed transactions. Need a coordinator to manage messages from participants
- 2PC can become a performance bottleneck

Additional Material

Overview:

- 2PC notation from the Book
- Terminology used by messages different, but essentially the protocol is the same
- Pointers to 3PC (fully described in the book)

Two-Phase Commit (1)



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

- Coordinator/Participant can be blocked in 3 states:
  - Participant: Waiting in INIT state for VOTE_REQUEST
  - Coordinator: Blocked in WAIT state, listening for votes
  - Participant: blocked in READY state, waiting for global vote

Two-Phase Commit (2)

- What if a “READY” participant does not receive the global commit? Can’t just abort => figure out what message a co-ordinator may have sent.
- Approach: ask other partcipants
  - Take actions on response on any of the participants
  - E.g. P is in READY state, asks other “Q” participants

State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

What happens if everyone is in ”READY” state?

Two-Phase Commit (3)

- For recovery, must save state to persistent storage (e.g. log), to restart/recover after failure.
  - Participant (INIT): Safe to local abort, inform Coordinator
  - Participant (READY): Contact others
  - Coordinator (WAIT): Retransmit VOTE_REQ
  - Coordinator (WAIT/Decision): Retransmit VOTE_COMMIT

2PC: Actions by Coordinator

Actions by coordinator:

```
write 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;
}
```

Why do we have the ”write to LOG” statements?

2PC: Actions by Participant

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;
}
```

- Wait for REQUESTS
- If Decision to COMMIT
  LOG=>Send=> Wait
- No response?
  Ask others
- If global decision,
  COMMIT OR ABORT
- Else, Local Decision
  LOG => send

2PC: Handling Decision Request

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 */
}
```

(b)

Note, participant can only help others if it has reached a global decision and committed it to its log.

What if everyone has received VOTE_REQ, and Co-ordinator crashes?