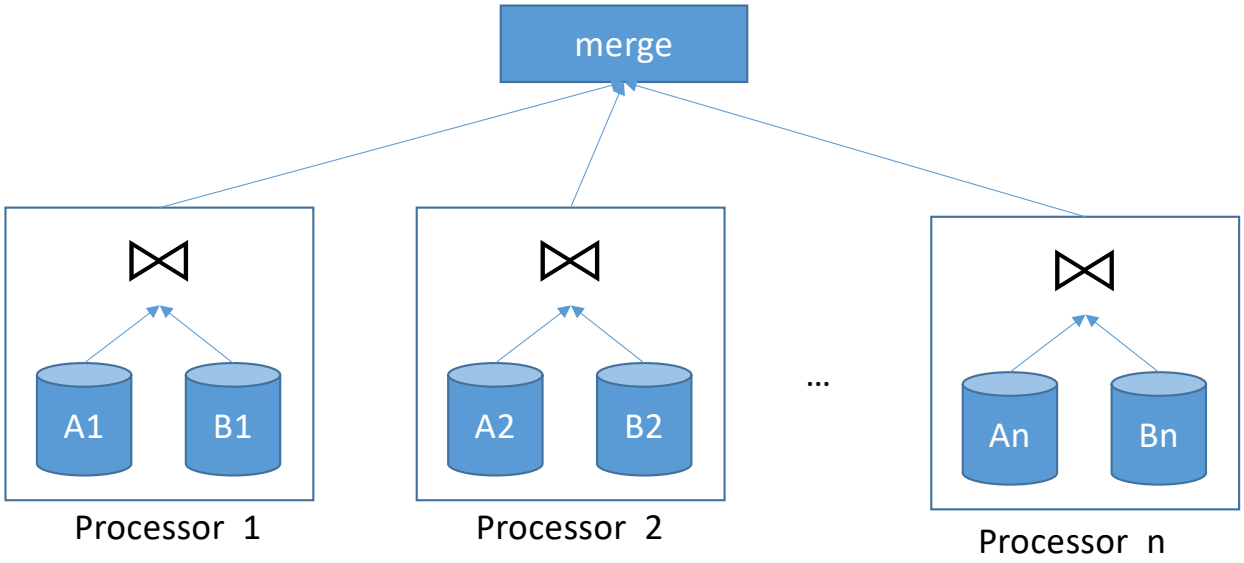
Lecture 17: Distributed Transactions & Two-Phase Commit

"The Arnolfini Wedding", Jan van Eyck, 1434

"two phase commit distributed transaction in early netherlandish style"
Stable Diffusion, November 8, 2022 at 1:52PM

Parallel DB Recap

- Last time: discussed parallel query execution
- Focused on *partitioned parallelism*



Partitioning Strategies

- Random / Round Robin
  - Evenly distributes data (no skew)
  - Requires us to repartition for joins
- Range partitioning
  - Allows us to perform joins without repartitioning, when tables are partitioned on join attributes
  - Subject to skew
- Hash partitioning
  - Allows us to perform joins without repartitioning, when tables are partitioned on join attributes
  - Only subject to skew when there are many duplicate values

Parallel Operations in a Partitioned DB

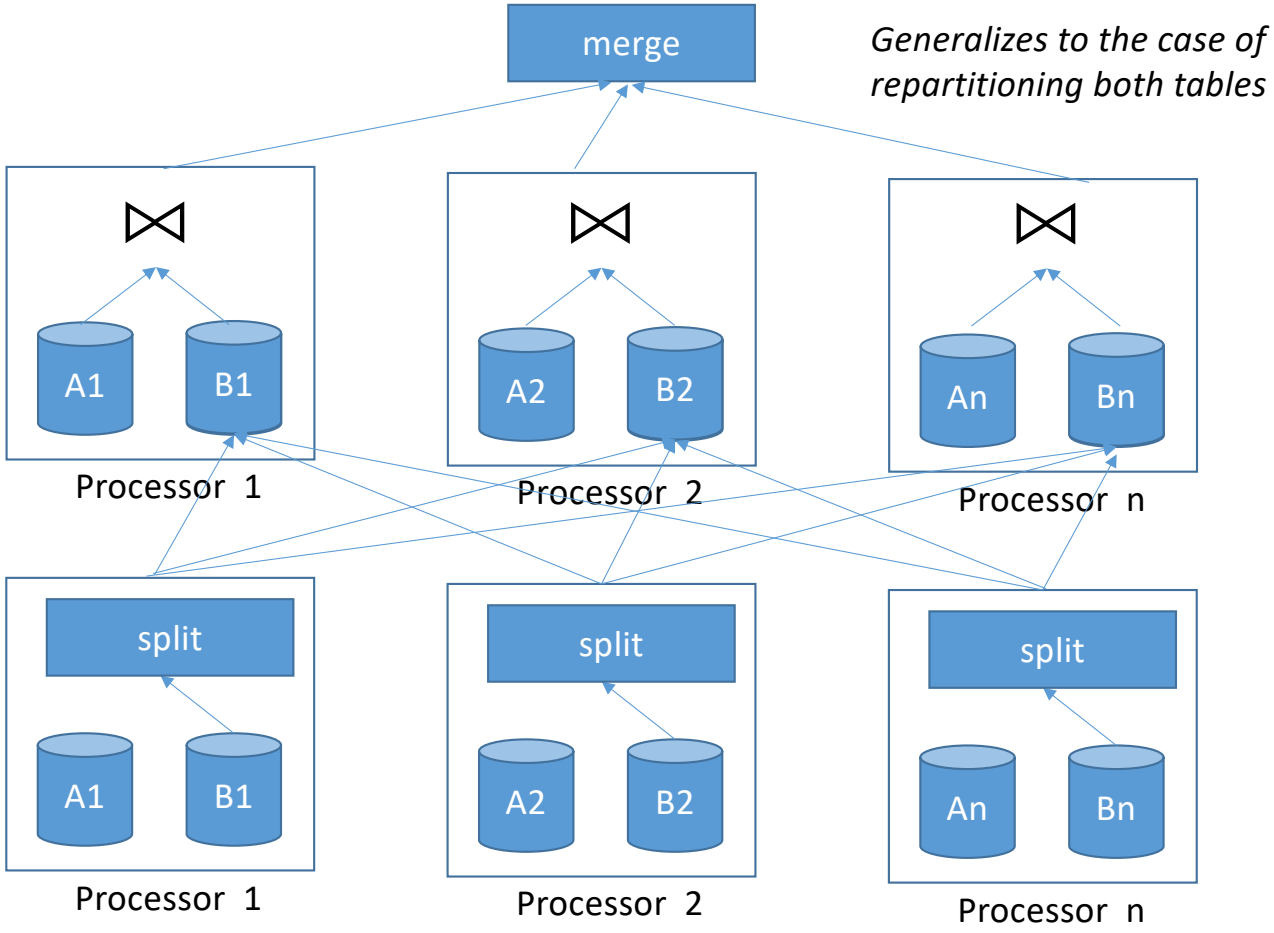
- SELECT
  - Trivial to “push down” to each worker
  - Depending on partitioning attribute, may be able to skip some partitions
- PROJECT
  - Assuming all columns are on each node, nothing to be done
- JOIN
  - Depending on data partitioning, may be able to process partitions individually and then merge, or may need to repartition
- AGGREGATE
  - Partially aggregate data at each node, merge final result

Join Strategies

- If tables are partitioned on same attribute, just run local joins
  - Also, if one table is replicated, no need to join
- Otherwise, several options:
  1. Collect all tables at one node
     - Inferior except in extreme cases, i.e., very small tables
  2. Re-partition one or both tables – "shuffle join"
     - Depending on initial partitioning
  3. Replicate (smaller) table on all nodes

Repartitioning Example

- Suppose A pre-partitioned on a, but B needs to be repartitioned



Distributed Transactions

- Suppose we have data on separate machines and want to run a transaction across them

- Example 1: reserve a rental car and an airline flight, and only commit if both are available.
- Example 2: transfer money from bank 1 to bank 2
- Example 3: add a friend to a social media graph, where user 1 is on Asia server and user 2 is on US server

Problem with Distributed Transactions

- Consider:

  BEGIN
  INSERT A → Machine 1
  INSERT B → Machine 2
  INSERT C → Machine 3
  COMMIT

  Problem: Machine 1 & 2 commit, Machine 3 crashes
  What happens?

Goal: Atomicity

- If one machine crashes, system should still preserve atomicity

➔ Crashing machine should recover & commit

or

➔ All machines (including crashing one) should rollback

In single-node system, a transaction is committed the moment the commit record goes to disk

In multi-node system, can't ensure commit record is all-or-nothing across all nodes!

Two-Phase Commit

- Key Idea: Add a new state, “PREPARED” to transactions

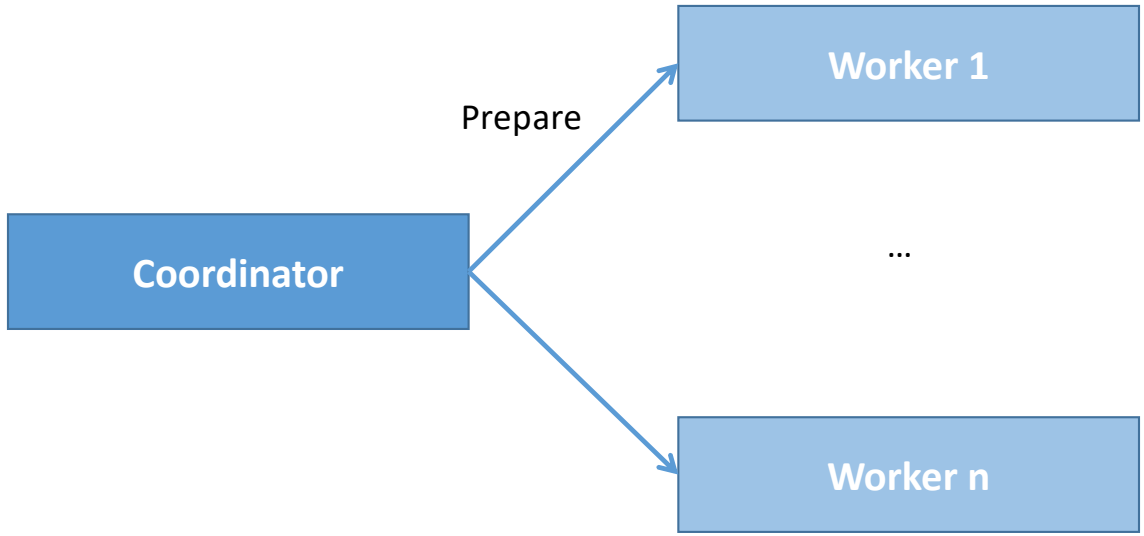
- Indicates that a node has done the work for a transaction, and the decision to COMMIT/ABORT will be done by a *coordinator*

- Once prepared, a node will not COMMIT or ABORT on its own

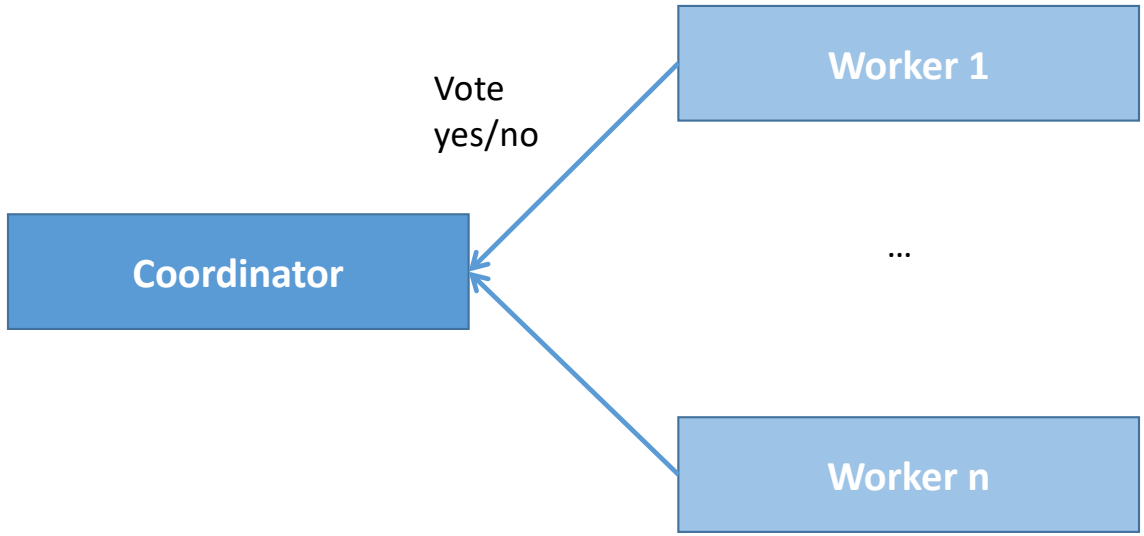
➔ ”Prepared” state must survive crashes

(Postgres Demo)

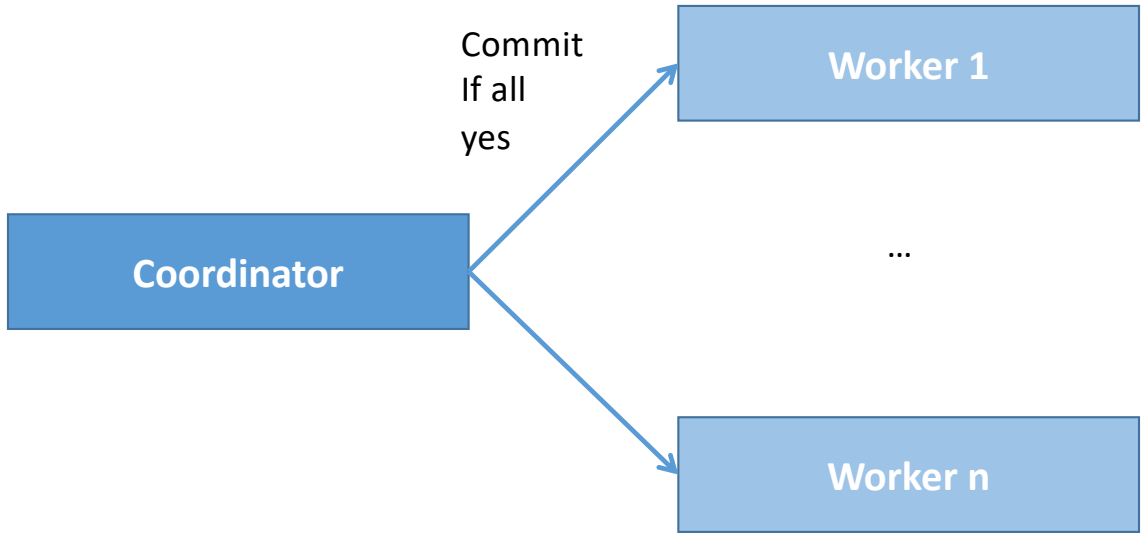
2PC Architecture



2PC Architecture

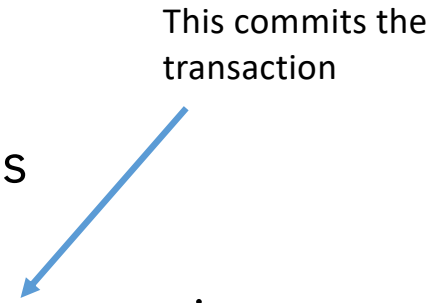


2PC Architecture



Two-Phase Coordinator Overview

1. Log start of transaction
2. Execute transaction on *worker* nodes
3. PREPARE each worker
4. Once workers are all prepared, log transaction commit
5. Commit each worker
6. Log DONE, so we know all transactions are done



(If one of the workers fails to prepare, abort each worker)

Coordinator Code

```
all_prepared = True
#log START, get TID
tid = logger.start_coord_txn()
for worker in workers:
    logger.start_worker_txn(worker, tid)
    do_work(worker)
    result = logger.prepare(worker, tid)
    all_prepared = all_prepared & result

if (all_prepared):
    logger.log(tid,"COMMIT")
else:
    logger.log(tid,"ABORT")

for worker in workers:
    if (all_prepared):
        logger.commit(worker, tid)
    else:
        logger.abort(worker, tid)

logger.log(tid,"DONE")
```

Logger:

```
def start_coord_txn(self):
    cur_tid = self.tid
    self.log(cur_tid, 'START')
    self.tid = self.tid + 1
    return cur_tid

def start_worker_txn(self, cursor, tid):
    cursor.execute("BEGIN TRANSACTION")

def prepare(self, cursor, tid):
    try:
        cursor.execute ("prepare transaction
                '%s'"%(t_name%(tid)))
        return True
    except psycopg2.DatabaseError as error:
        return False

def commit(self, cursor, tid):
   cursor.execute ("commit prepared
                '%s'"%(t_name%(tid)))
```

What If Coordinator Crashes

- Log tells us which transactions were running
- If before Coordinator COMMIT, all workers should abort
  - Some may have prepared, some may not
  - (Workers may be asked to abort unprepared transactions)
- If after Coordinator COMMIT, but not DONE all workers should commit
  - Some may have committed, some may not
  - (Workers must be asked to commit transactions again)

Recovery Code

```
def recover(self):
    to_abort = []
    to_commit = []
    max_tid = 0
    for (tid,cmd) in self.log_lines():
        if cmd == 'START':
            to_abort.append(tid)
            max_tid = max(self.tid, tid)
        if cmd == 'COMMIT':
            to_abort.remove(tid)
            to_commit.append(tid)
#txns logged as 'ABORT' implicitly abort
        if cmd == 'DONE':
            if tid in to_abort:
                to_abort.remove(tid)
            if tid in to_commit:
                to_commit.remove(tid)
```

```
if (len(to_abort) > 0 or
    len(to_commit) > 0):
    workers = self.get_workers()
    for txn in to_abort:
        for worker in workers:
            self.abort(worker,txn)
        self.log(txn,'DONE')
    for txn in to_commit:
        for worker in workers:
            self.commit(worker,txn)
        self.log(txn,'DONE')
    for worker in workers:
        worker.close()
self.tid = max_tid + 1
```

What Happens in Worker PREPARE?

- Because PREPAREd state is not committed, a worker must:
  - Hold locks until COMMIT or ABORT
  - Be able to COMMIT / ABORT even if it crashes

- Because PREPAREd state must survive a crash, a worker must
  - Log that it is prepared (before acknowledging the prepare to coord)
  - Recover back into the PREPAREd state (re-acquiring locks!)

- Requires logging locked objects, and forcing log to disk before acknowledging PREPARE

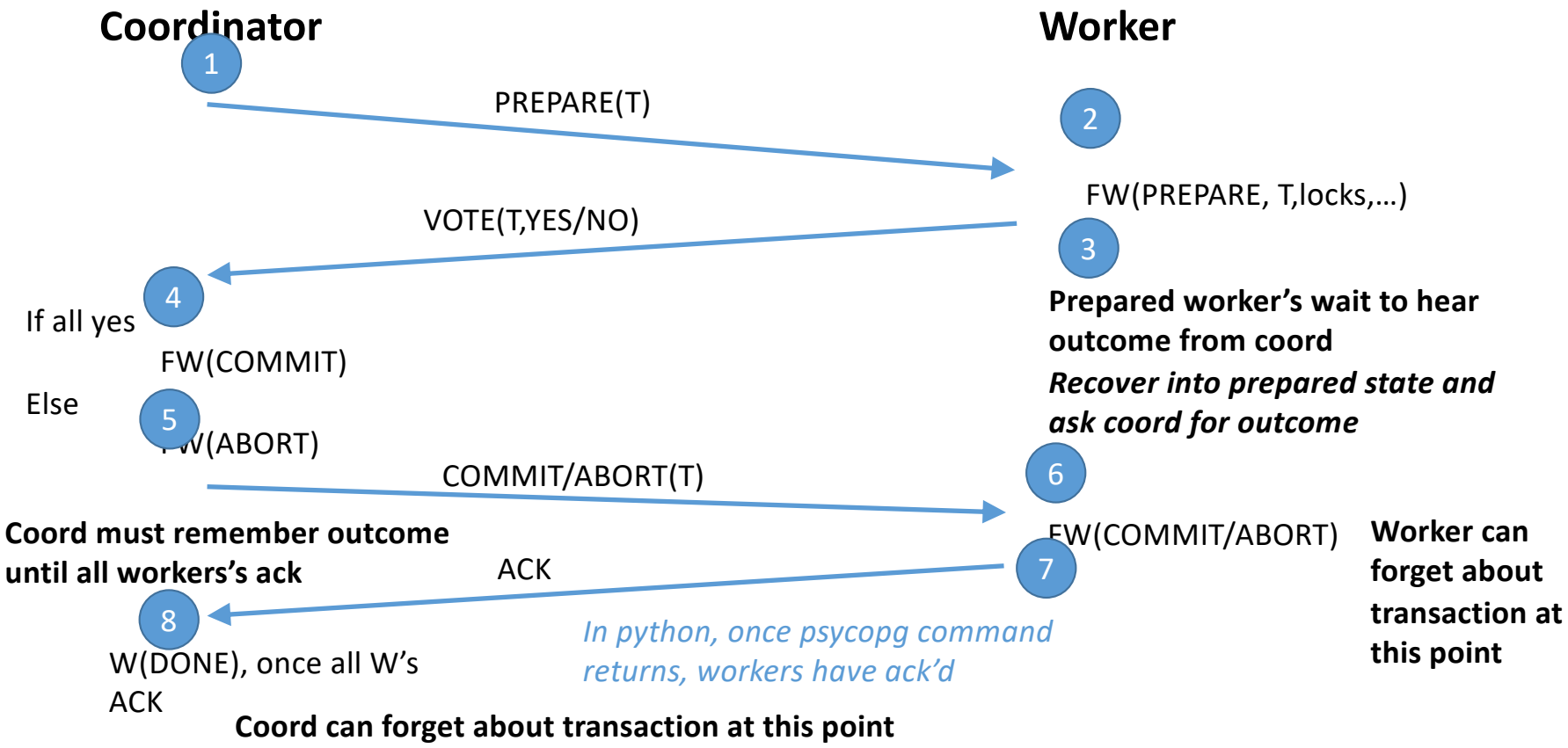
Worker Recovery Process

- Each worker has a *recovery process* that keeps track of the outcome of transactions running on the site
- If a site is prepared and crashes, it needs to ask coordinator about the outcome of the transaction on recovery

- This is not handled in our pseudocode, or Postgres
- Would require a separate monitor for each DB

Two-phase commit protocol

FW = Force Write

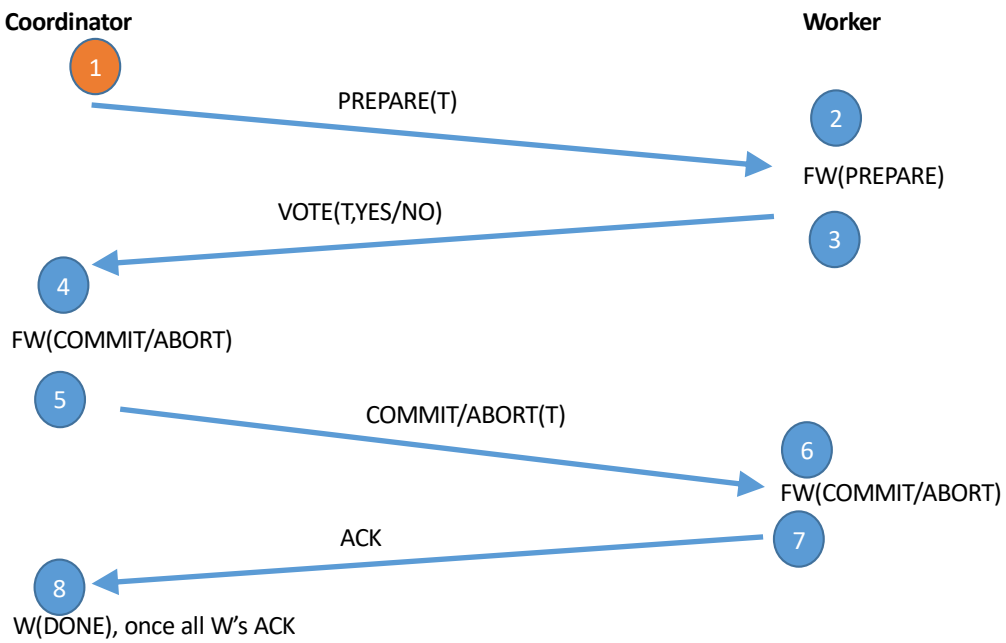


Failure Cases

(1) Coordinator crashes before sending PREPARE

- Coord – will recover, abort transaction just as in normal recovery, discarding all state

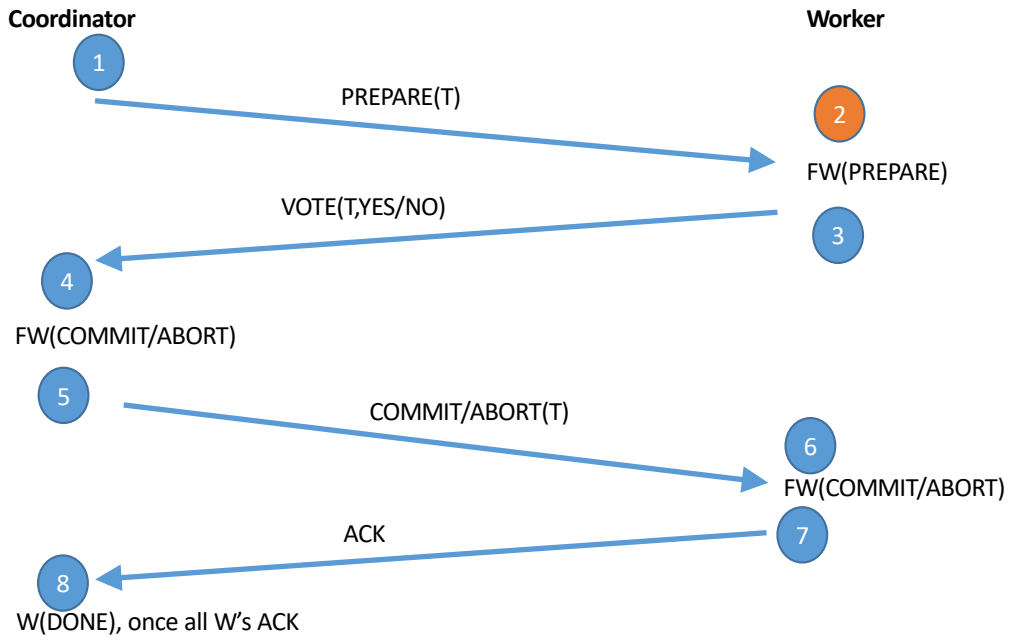
- Worker – can safely abort;
  - Add *recovery process* that polls coordinator for status of outstanding txns
  - *Coord*, which has no record of txn, will tell worker to abort



Failure Cases

(2) Worker crashes before receiving PREPARE

- Coord – will never hear reply, will abort
- Worker – will recover, rollback txn during recovery



Failure Cases

(3) Worker crashes after PREPARE

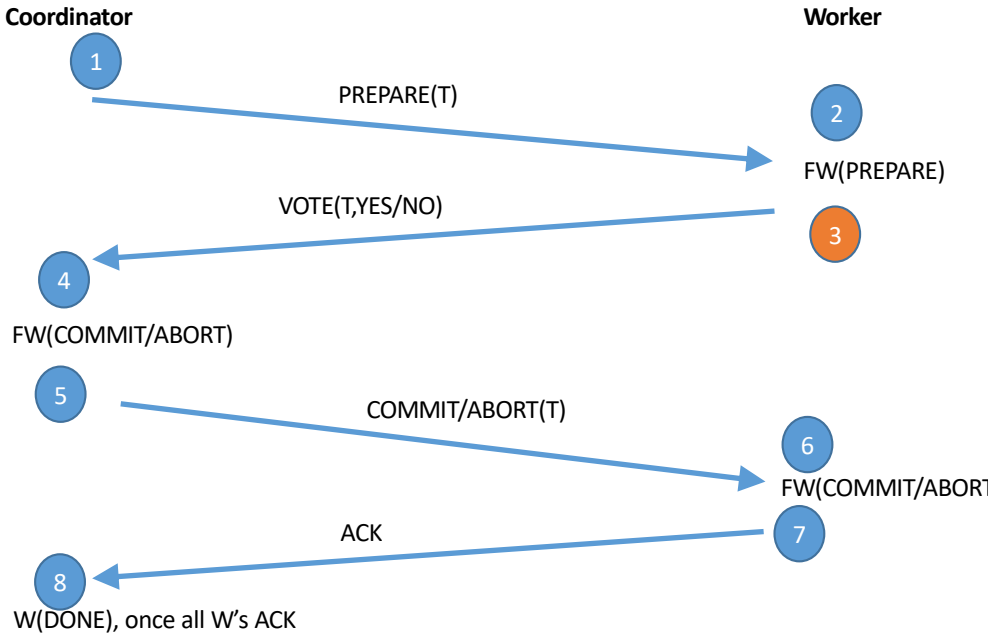
Must determine outcome from coord:

Two cases

(a) It already sent its vote, and coord is waiting for an ack -- thus, worker can learn fate by contacting coord

(b) It didn't send its vote, in which case coord may or may not have timed out.

- If it has not timed out, it can vote.
- If it has timed out, it *must* have aborted, and will tell the worker this.

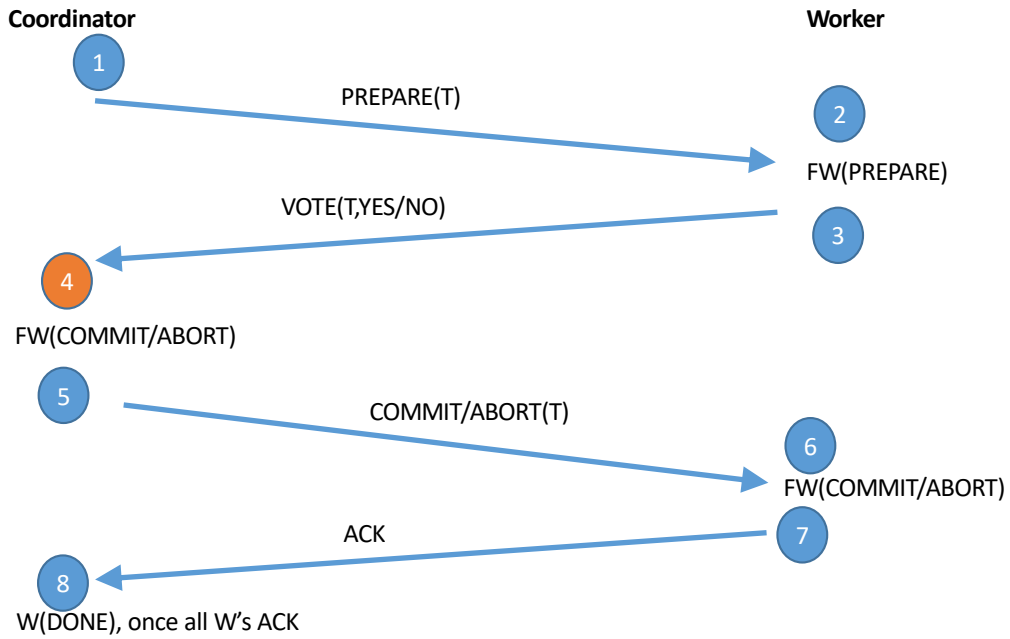


Failure Cases

(4) Coordinator crashes before receiving all votes

Coord aborts during recovery, informs workers

Note that workers who have prepared must wait for coordinator to restart to hear outcome

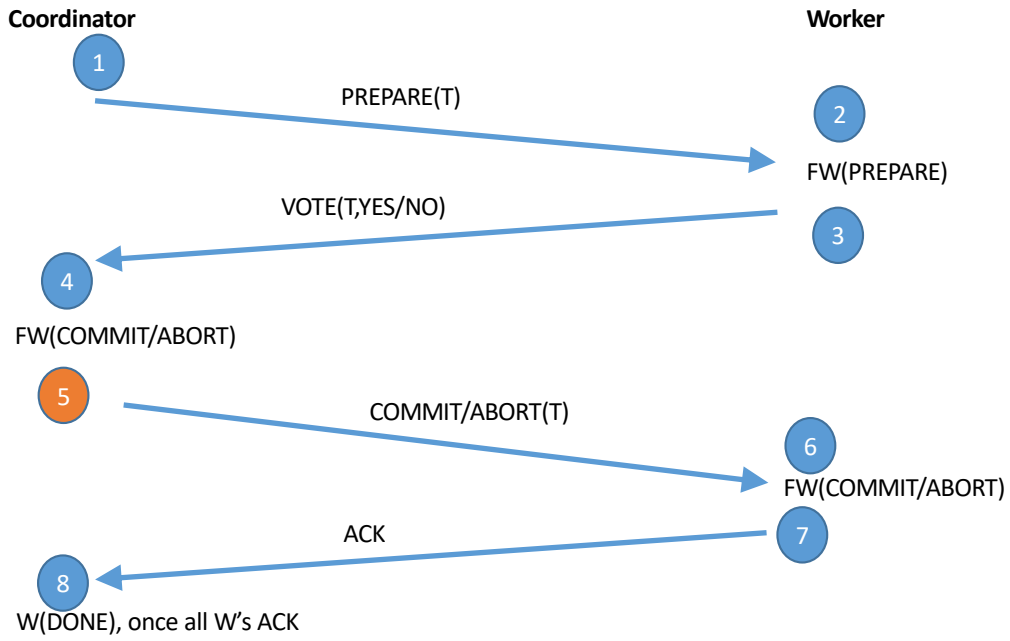


Failure Cases

(5) Coordinator crashes after writing COMMIT

No DONE record; coordinator sends commits to all workers

Workers must wait to hear outcome

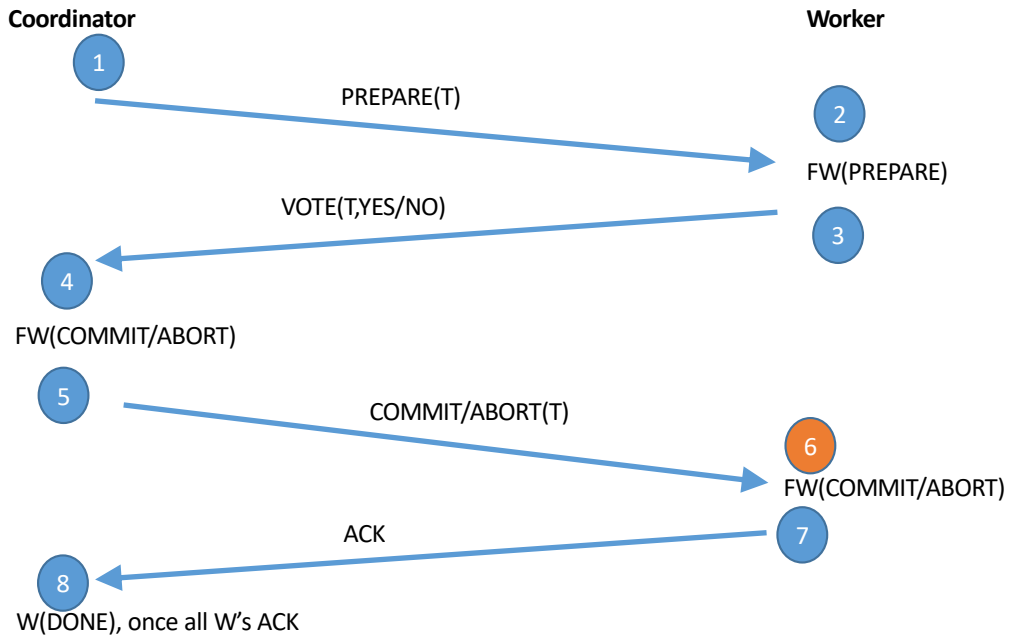


Failure Cases

(6) Worker crashes before receiving COMMIT / ABORT

Upon recovery, recovery process polls for outcome.

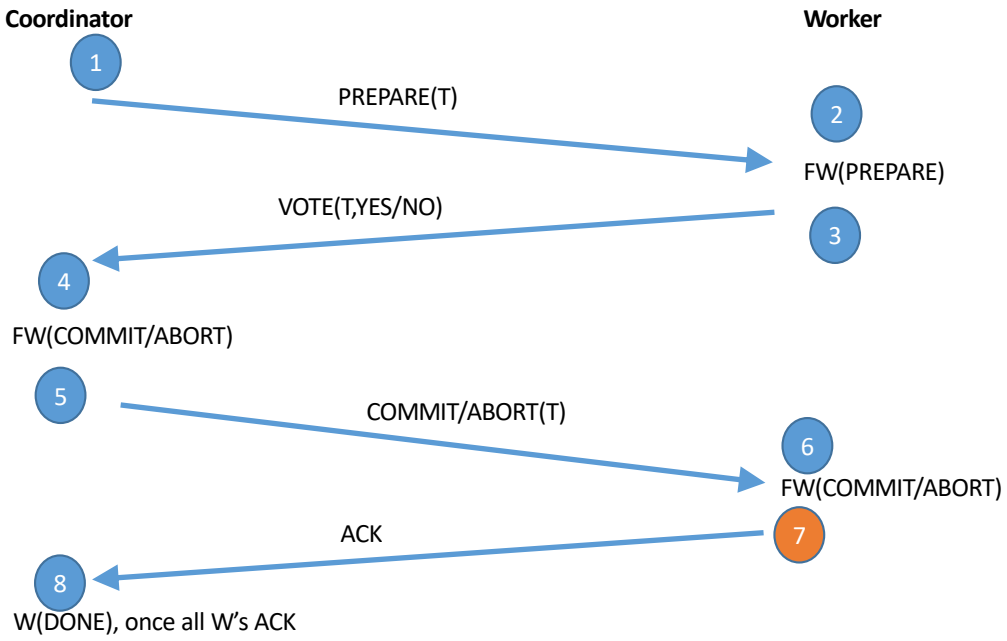
Since coordinator has not received ACK, it still knows state.



Failure Cases

(7) Worker crashed after writing COMMIT record, before ACKing.

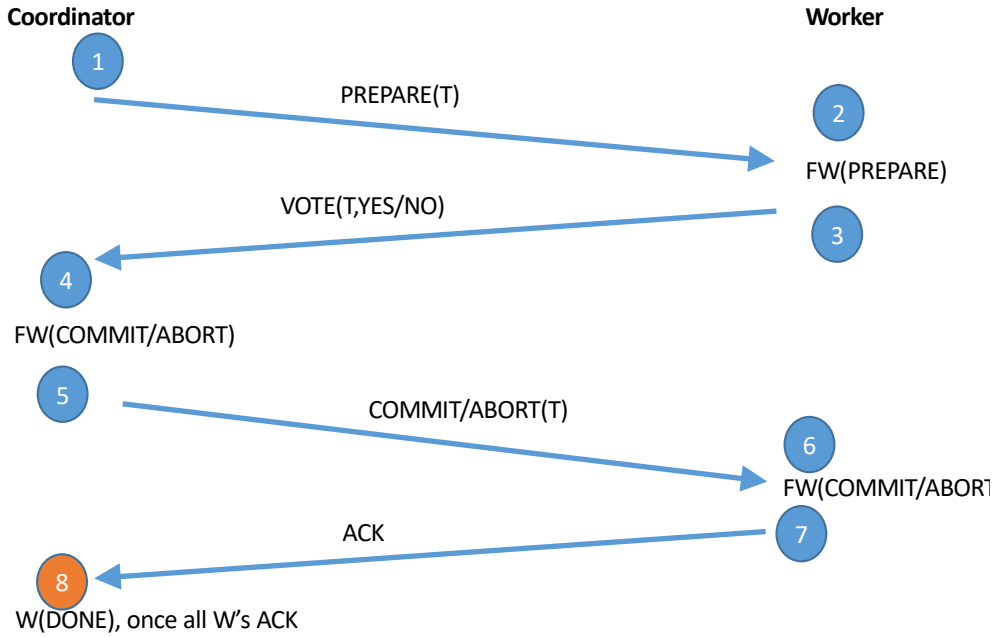
Worker will recover, transaction will be committed. Coordinator will periodically send a COMMIT message, which worker will ACK without writing any additional state.



Failure Cases

(8) COORD Crashed after receiving some ACKs.

COORD will send COMMIT/ABORT to all workers, who will ACK.



Read-only Workers

- If a worker is read-only (RO), it can send a "READ VOTE"
  - Doesn't need to write any log records
  - Can forget the transaction after it votes

- Coord doesn't need to send ABORT/COMMIT to RO workers

- If all workers are RO, no ABORT/COMMIT messages needed

Two Variants

- Presumed Abort and Presumed Commit
- Avoid some logging when transactions abort / commit

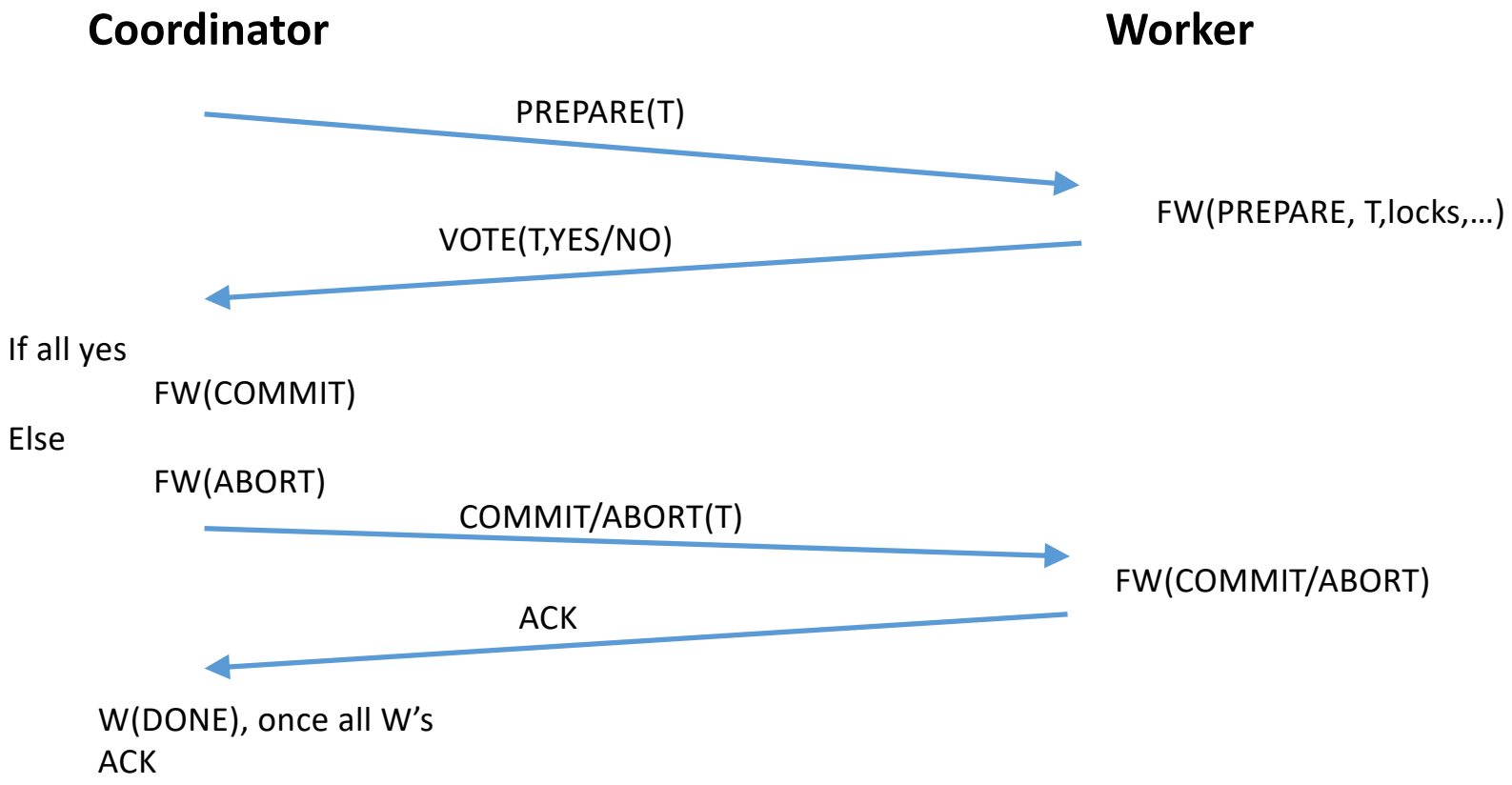
Presumed Abort

- Notice that in the existing protocol, if a recovery process contacts coordinator, and coordinator has no info about transactions, it replies “abort”

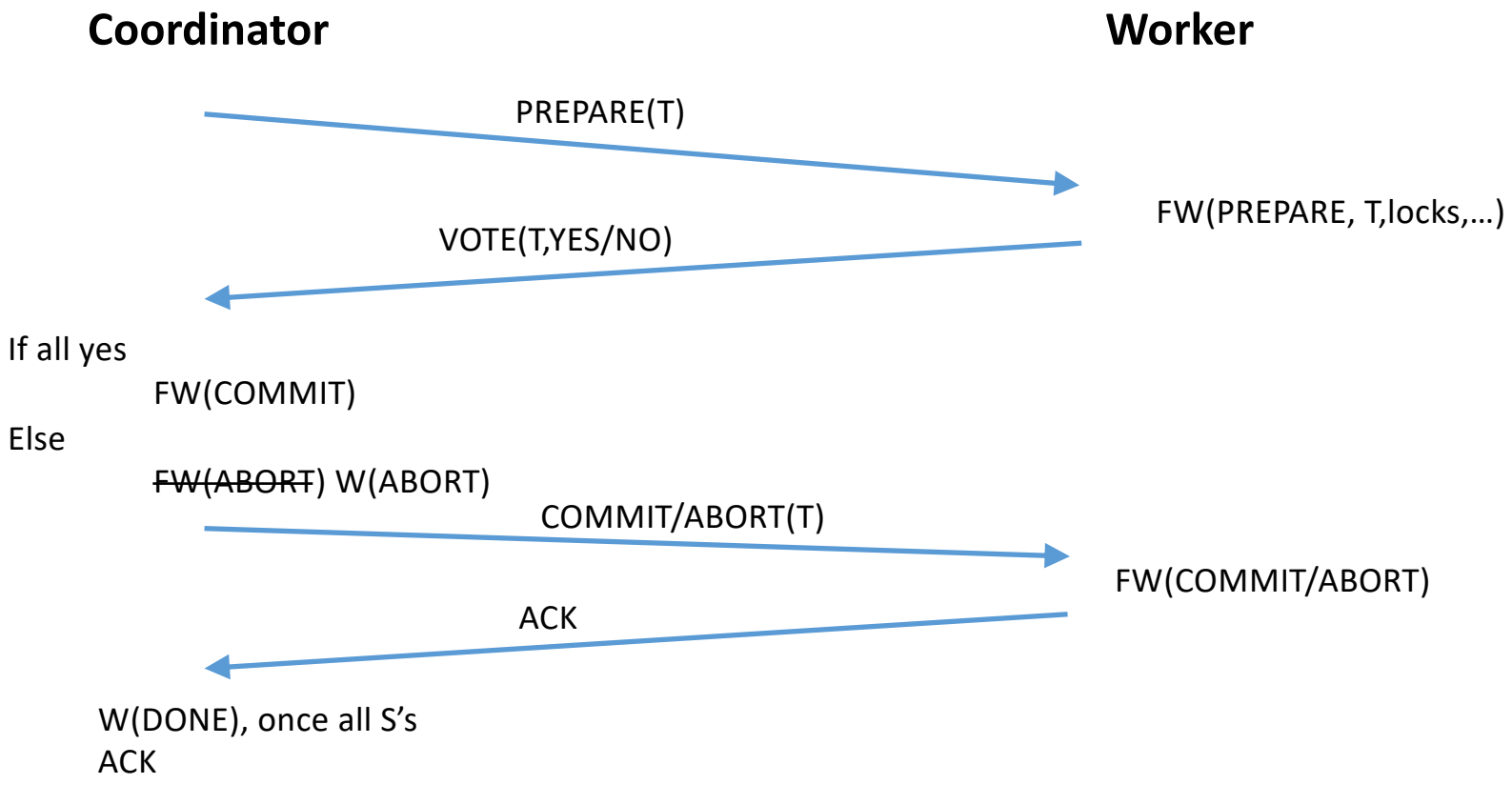
- Implies we do not need to force writes for aborting transactions

- Committing transactions are unchanged

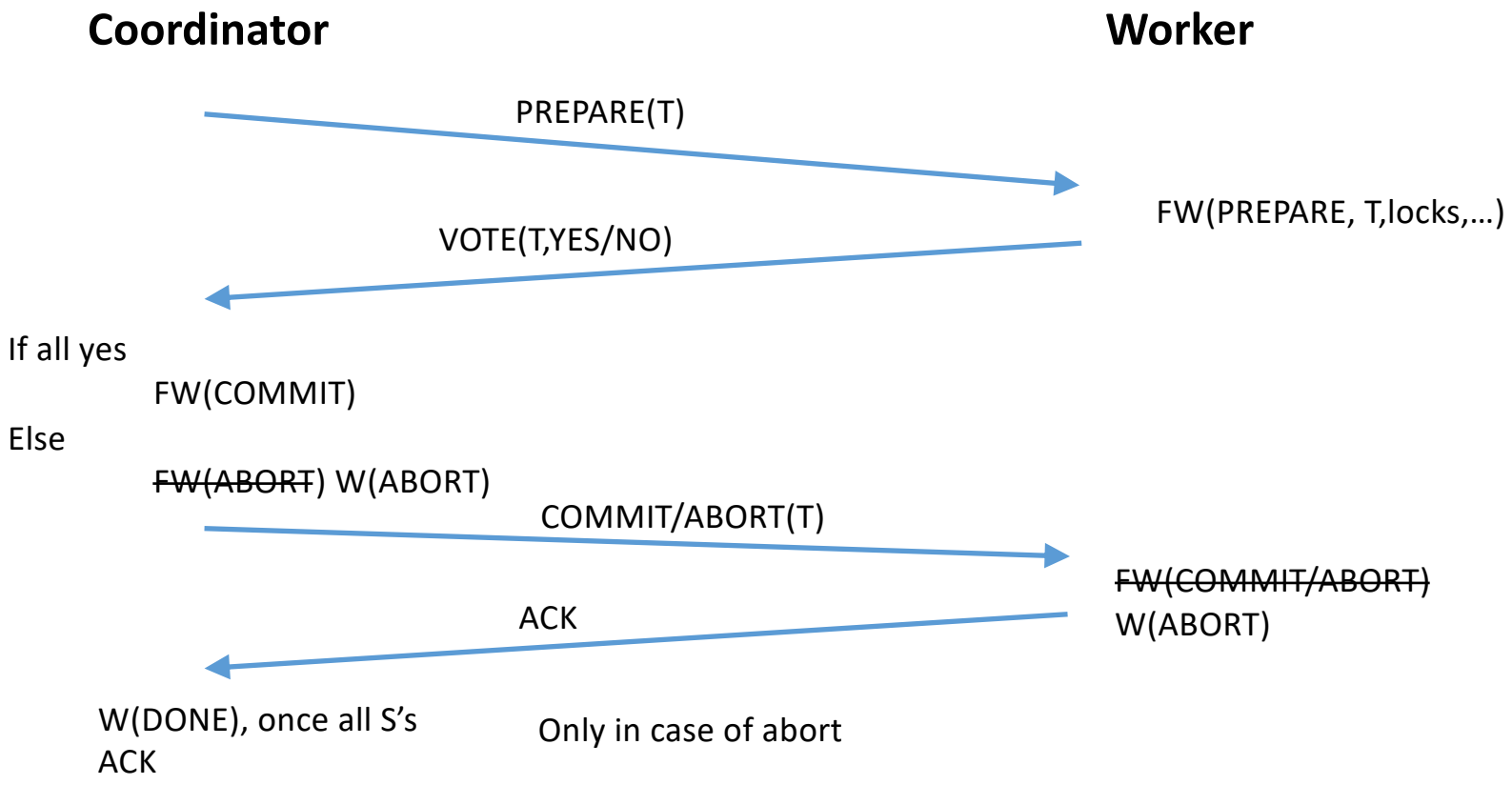
Presumed Abort – if transaction aborts



Presumed Abort – if transaction aborts



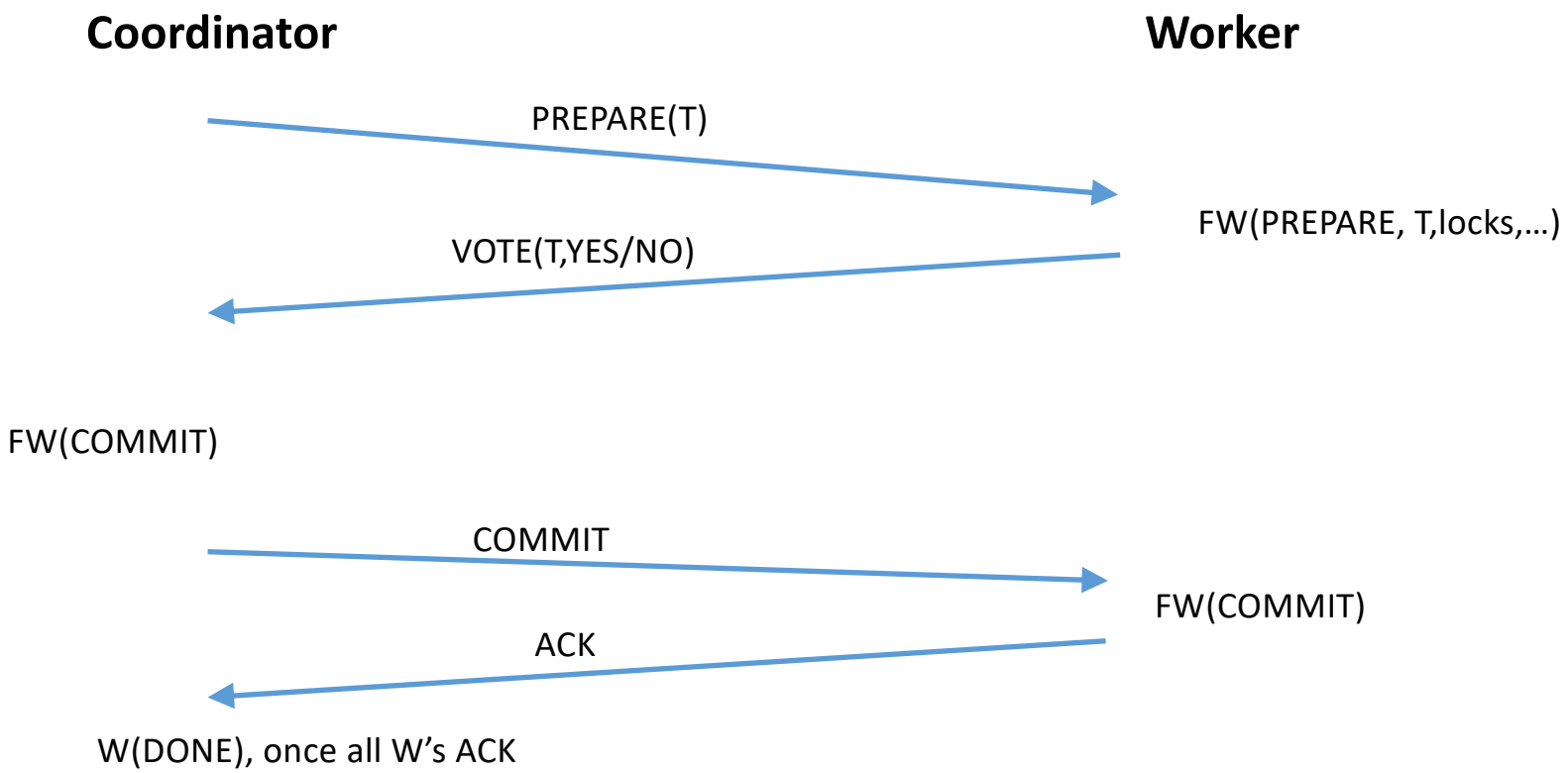
Presumed Abort – if transaction aborts



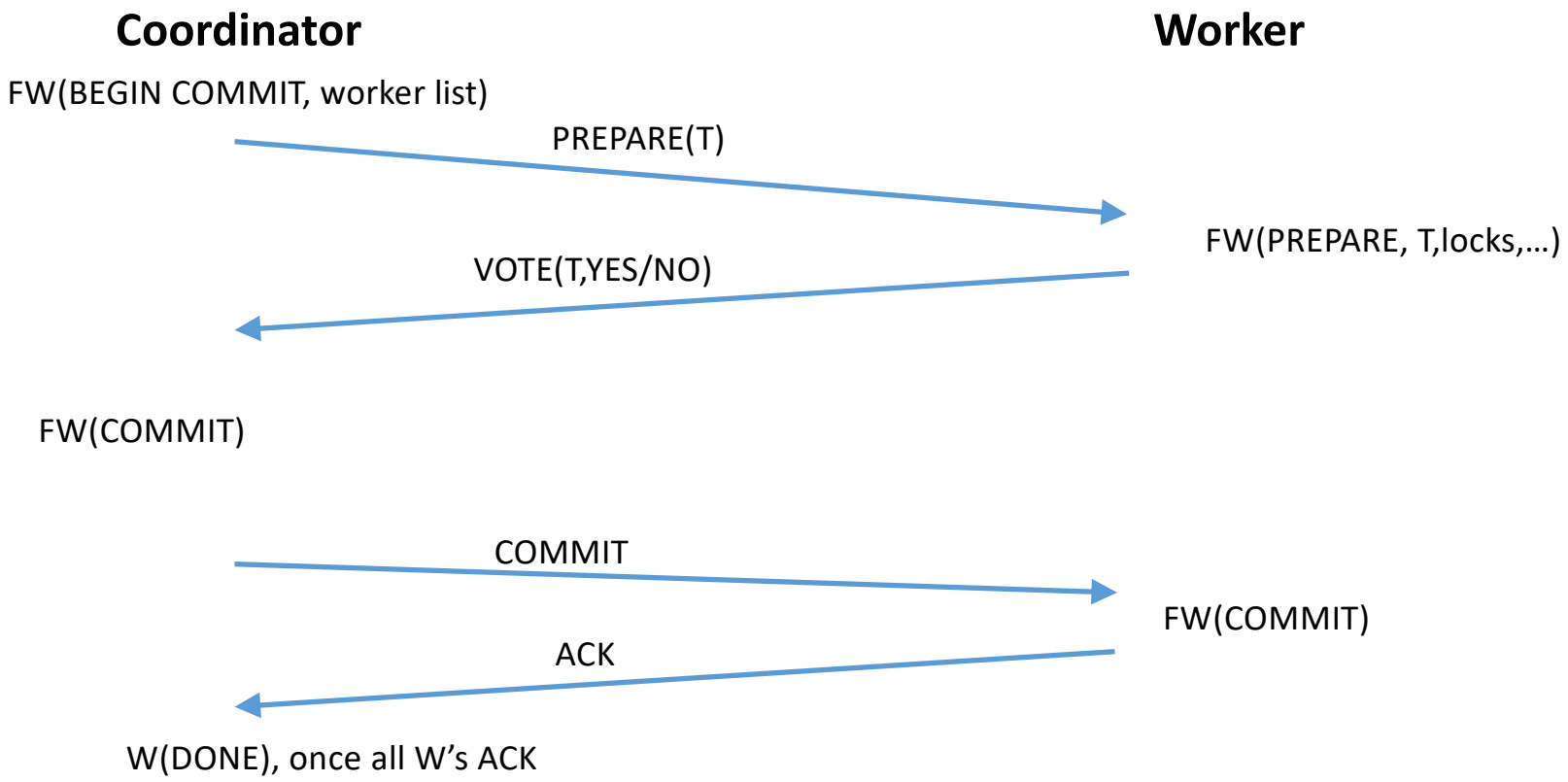
Presumed commit – if transaction commits

- Can't just reply "COMMIT" in no information case
  - Suppose coord sends prepare messages, then crashes
  - Worker sends vote, doesn't hear anything, re-requests
  - Eventually coord recovers, rolls back, and replies "COMMIT" (because it has no information about txn)
- Soln: prior to sending prepare, coord force writes an additional "BEGIN COMMIT" records with a list of all workers
  - If it crashes prior to writing COMMIT/ABORT, it can restart commit process, contact workers, collecting votes and sending outcomes
- Adds an additional write on coord, but allows worker COMMIT to be an async write

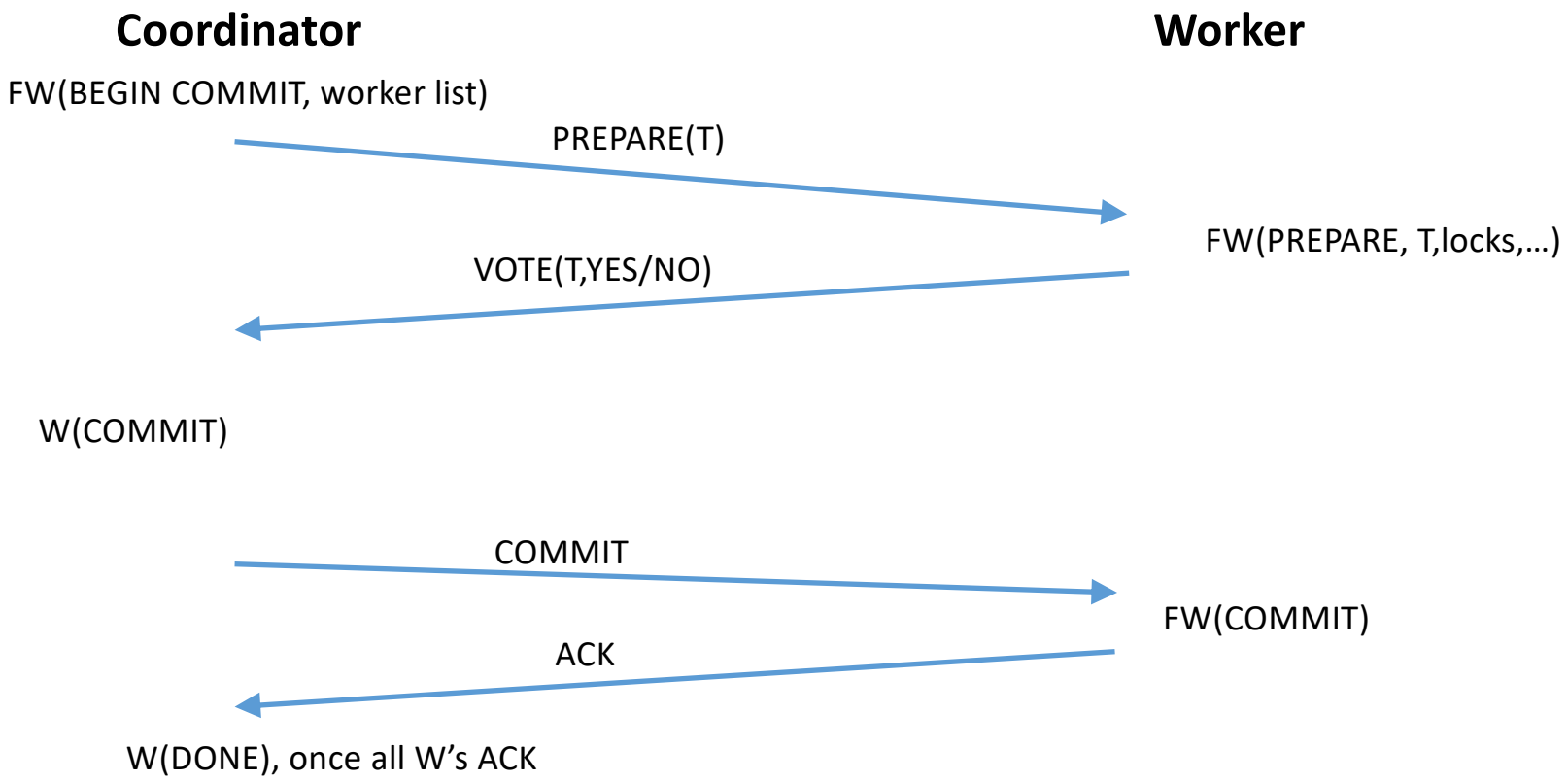
Presumed commit – if transaction commits



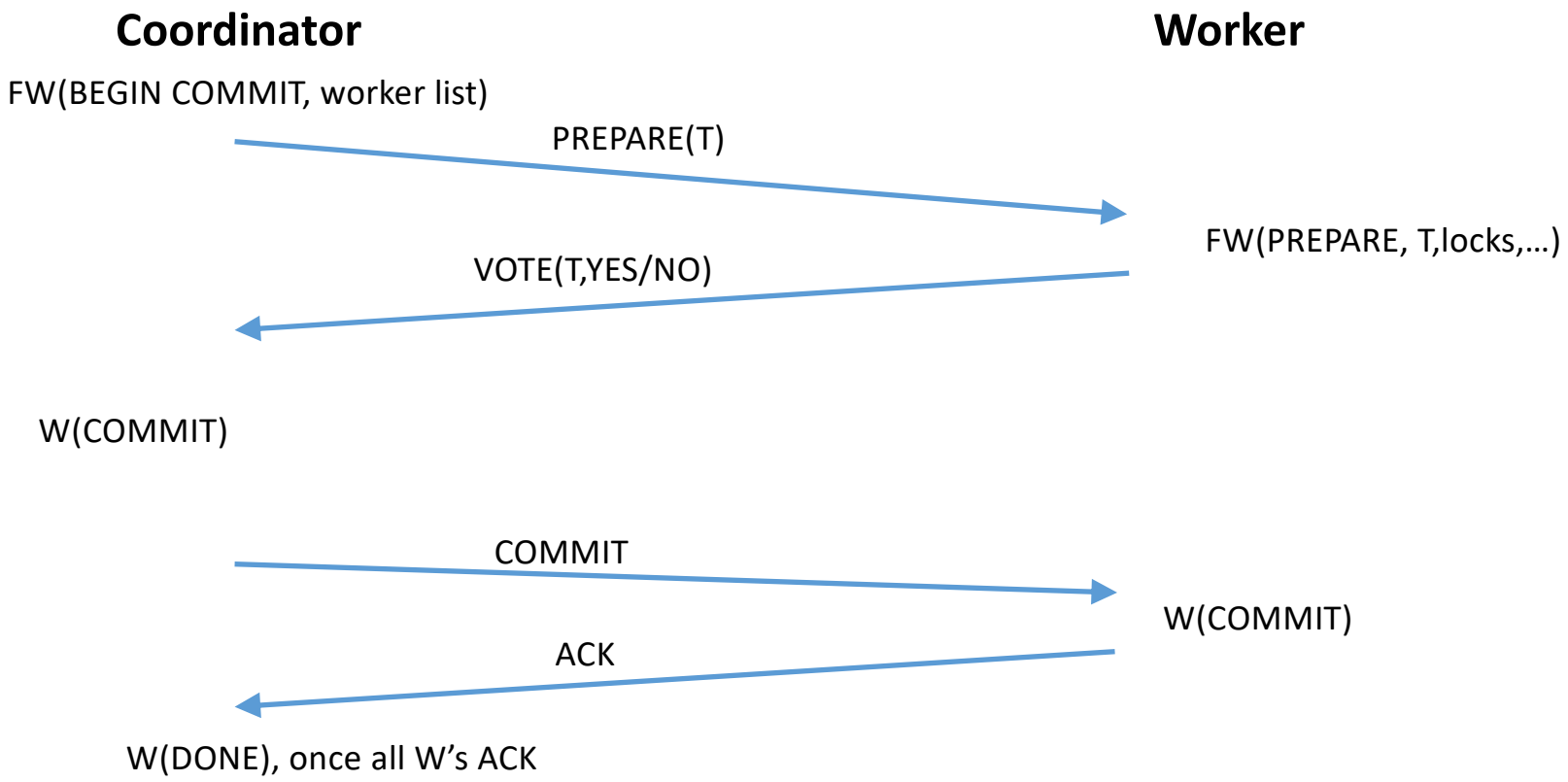
Presumed commit – if transaction commits



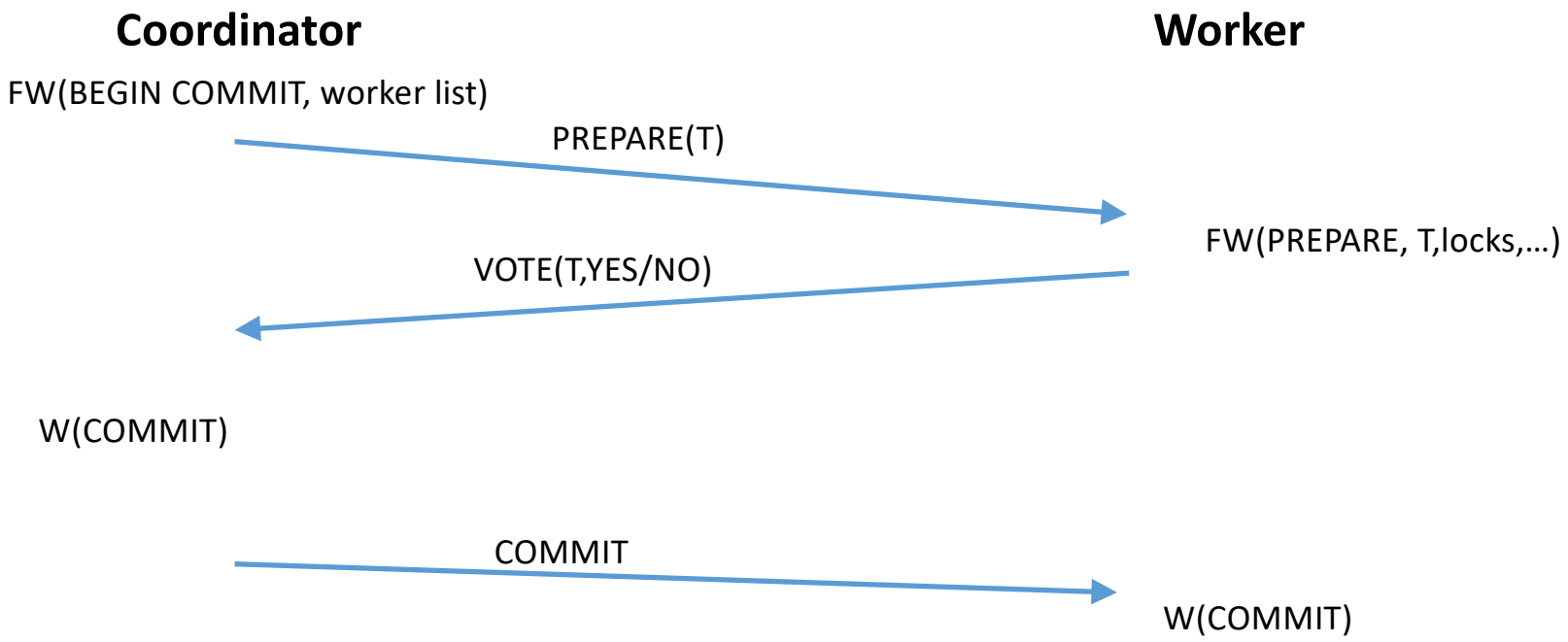
Presumed commit – if transaction commits



Presumed commit – if transaction commits



Presumed commit – if transaction commits



Abort case still retains all writes
of regular protocol

Summary: Write/Message Complexity

W = Write
F = Force Write
M = Message

Messages for committing transaction

	Coord Update or Read-only	**Worker Update**	**Read-Only**
Standard	2W,1F,1M(R/O),2M(U)	2W,2F,2M	0W,0F,1M
PA	2W,1F,1M(R/O),2M(U)	2W,2F,2M	0W,0F,1M
PC	2W,1F,1M(R/O),2M(U)	2W,1F,1M	0W,0F,1M

PA only helps in abort cases (not this one)
PC costs more writes on coord, but has fewer writes on workers

2PC – Problems

- 2 network round trips + synchronous logging ➔ high overheads
  - Particularly when Coord and Worker are far apart, i.e., in different data centers
- If Coord fails, Workers must wait, or somehow choose a new coordinator
- If Coord + 1 Worker fail, no way to recover
  - Coord may have told failed Worker about outcome, it may have exposed results

2PC sacrifices *availability* of system for *consistency*

2PC is probably not a good choice in a wide-area distributed setting

Due to possibility of network failures, and wide area latency

Alternatives: use a more complicated consensus protocol (e.g., Paxos), use deterministic execution, or relax consistency