Fault Tolerance

Atomic Commitment

Pedro F. Souto (pfs@fe.up.pt)

May 2, 2014

Atomic Commitment: Informal

Problem How to ensure that a set of operations executed in different processors?

- either are **all** executed (**all committed**)
- or **none** of them is executed (**all aborted**)

Observation The origin of this problem is distributed databases, i.e. distributed transactions:

- A transaction comprises operations (sub-transactions) in different DBs
- A transaction must be **atomic** (and also CID)

Observation AC is useful:

- Not only when processes may fail
- But also when the operations may not be performed because of some reason other than the failure of the process that execute them
  - E.g., because of a deadlock in one of the sub-transactions

Transaction's ACID[1] Properties (Reminder)

- A **tomicity:** either all operations of a transaction are executed or none of them
- C **onsistency:** a transaction transforms a consistent state into another consistent state
- I **solation:** the effects of a transaction are as if no other transactions executed concurrently
- D **urability:** the effects of a transaction that commits are permanent

[1] Coined by T. Häerder and A. Reuter in 1983

Transaction's Consistency Contract (R. Guerraoui)

System

- Atomicity
- Isolation
- Durability

Programmer

- Consistency (local)

Consistency (global)

Atomic Commitment: Formal

Def. Consider a set of *n* processes such that:

1. Each process has to decide one of two values: **commit**/**abort**
2. Each process shall vote/propose one of these two values
3. The value decided by each process must satisfy the following assertions:

   AC1 All processes that decide, must decide the same value

   AC2 The decision of a process is final (it cannot be changed)

   AC3 If some process decides **commit**, then all processes must have voted commit

   AC4 If all processes voted **commit** and there are no failures, then all processes must decide commit

   AC5 For any execution containing only failures that the algorithm is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes eventually reach a decision

Two-Phase Commit: A solution for AC (1/9)

Assumptions Processes may fail by crashing and recover

- Each process has storage whose content survives a crash

Outline The protocol has two kinds of processors:

Coordinator there is only one coordinator process, at any time instant

Participant every process that performs an operation is a participant

- The coordinator may also be a participant, in which case it will have to perform both the coordinator-side and the participant-side of the protocol

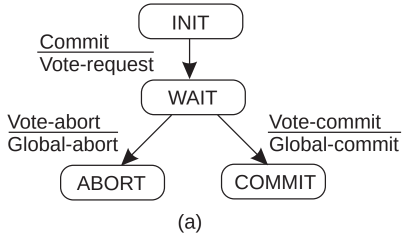
Two-Phase Commit: Basic Protocol (2/9)

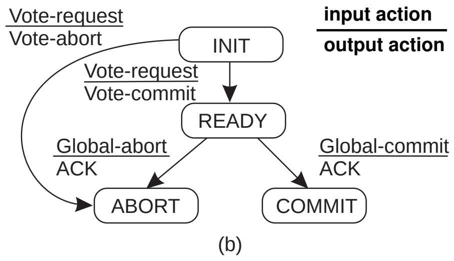
Outline The protocol has two phases:

- **Phase 1** Upon request from the application:
  - **Coordinator** sends a VOTE-REQUEST to each participant and waits for their reply
  - **Participant** upon receiving a VOTE-REQUEST each process sends its vote, either VOTE-COMMIT/YES or VOTE-ABORT/NO
- **Phase 2** Once the coordinator determines that is time to decide
  - **Coordinator** decides/sends:
    - **GLOBAL-COMMIT** if it received a VOTE-COMMIT/YES from all participants
    - **GLOBAL-ABORT** otherwise
  - **Participant** decides according to the message received from the coordinator

Two-Phase Commit: Simplified State Machine (3/9)

State Machines



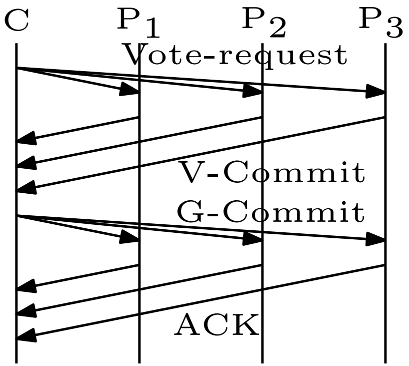


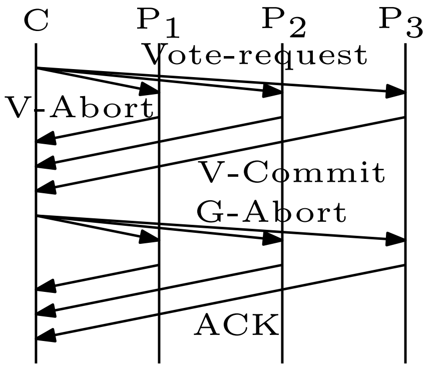
Coordinator

Participant

Two-Phase Commit: Fault-free Execution (4/9)

Possible Executions

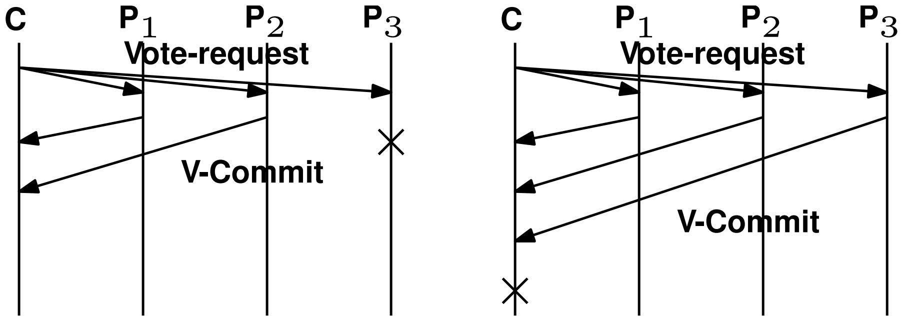




- In the absence of faults, the protocol is straightforward

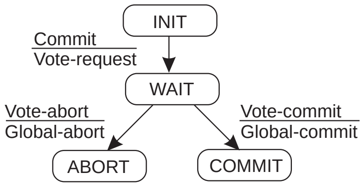
Two-Phase Commit: Faulty Executions (5/9)

Possible Executions with Faults

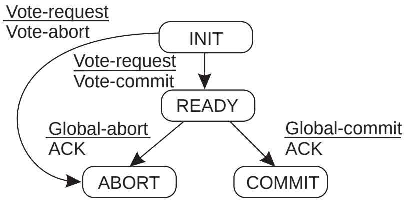


- We need to specify what to do in the case of failure
  - In practice, we use **timeouts** to detect failure

Two-Phase Commit: Timeouts (6/9)



(a)



(b)

Timeout Actions Actions taken by a process upon a timeout

Coordinator Waits for messages only in the WAIT state

- Decides ABORT and sends a GLOBAL-ABORT to participants

Participants Waits for messages both in:

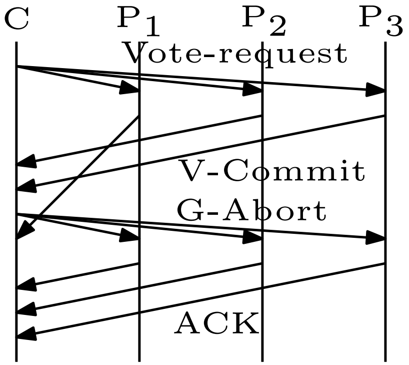
INIT Decide ABORT and move to corresponding state

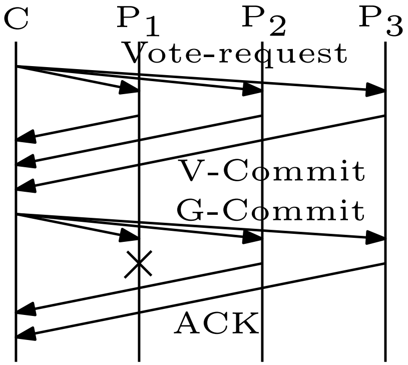
READY Must communicate with other processes to find out the outcome

- May have to wait until the coordinator recovers to find out what was the decision

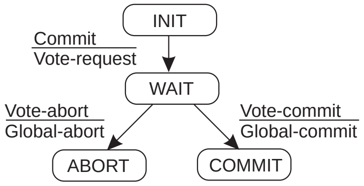
Two-Phase Commit: Execution with Faults (7/9)

Possible Executions with Faults

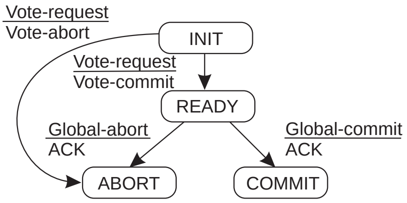




Two-Phase Commit: Recovery (8/9)



(a)



(b)

Recovery Actions Actions taken by a process upon recovery from a crash

- If process has not decided yet then
  - If crashed while waiting for a message, take the corresponding timeout action
  - Otherwise (coordinator in the INIT state) decide ABORT

Two-Phase Commit: Recovery (9/9)

- To allow **local** recovery, processes may have to write entries to a **log on disk**
  - The write of the entry should be performed before or after sending the corresponding messages?
  - In addition to the state of the protocol, the log may be used to store application data
- The 2-phase commit protocol satisfies assertions AC1-AC5, even in the presence:
  - communication faults, including partitions
  - non-byzantine node failures
- The main problem with this protocol is that it may require **participants** to block
  - This problem can be made less likely by using the three-phase commit protocol

Atomic Commit: Local Recovery and Blocking

Impossibility of independent recovery there is no AC protocol that always allows local recovery, i.e. without communication with other processes

- If a process is uncertain when it fails, ...

Non-blocking impossibility there is no AC protocol that never blocks in the presence of either:

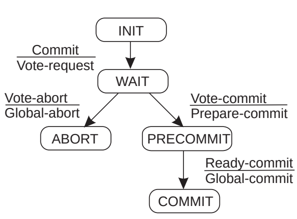
Failures on all sites or
Communication failures

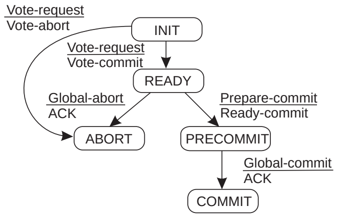
2 Phase-Commit may block even when there is no failure on all sites:

Can we do better?

Three-Phase Commit (1/2)

- Adds a phase between the 2 phases of the 2PC protocol, in which the coordinator reveals its intention to COMMIT


INIT
Commit / Vote-request
WAIT
Vote-abort / Global-abort
Vote-commit / Prepare-commit
ABORT
PRECOMMIT
Ready-commit / Global-commit
COMMIT
(a)


Vote-request / Vote-abort
INIT
Vote-request / Vote-commit
READY
Global-abort / ACK
Prepare-commit / Ready-commit
ABORT
PRECOMMIT
Global-commit / ACK
COMMIT
(b)

Coordinator

Participant

- The PRECOMMIT states ensure the **non-blocking** condition:
  - No process can commit while another process is in an uncertain state (INIT, WAIT, READY), i.e. can decide either way

    Note a process in the PRE-COMMIT state is not uncertain:
    - It will decide **commit**, unless it fails (in which case it may or not)
- It can be shown that this condition is necessary and sufficient to prevent blocking **unless ...**

Three-Phase Commit (2/2)

All processes fail

- **Assuming there are no communication failures**
  - If communication failures occur, AC1 (agreement) may be violated

There is no majority

- If a majority is able to communicate:
  - If a majority of the participants are in the READY state, they can decide ABORT
  - If a majority of the participants are in the PRECOMMIT state, they can decide PRECOMMIT
- Because two majorities must overlap, no different decisions can be made
  - Note that a process in the PRECOMMIT state may still ABORT, but only if the coordinator fails
- If there is no majority (as a result of network partition, e.g.) processes may block

In either case, we need appropriate timeout/recovery actions

- Anyway, virtually all systems implement 2PC, not 3PC

Stable Storage (1/2)

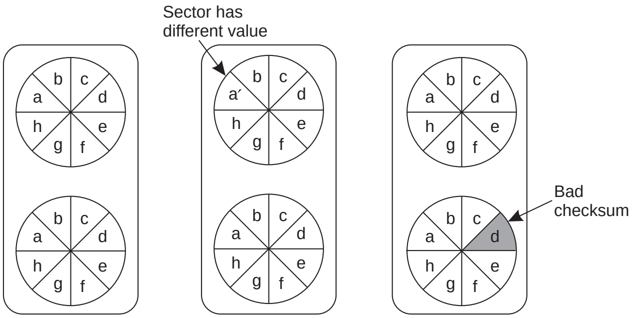
Problem

- Many protocols like 2PC assume that the state of processes, or at least some part of it, survives the failure of the process
- Usually, this means to store the state on disk
- How do you ensure that the data survives disk failures?

Solution: Stable Storage

- Use two identical disks
- Writing a block requires writing first in disk 1 and then in disk 2
- Upon reading a block, try disk 1 first, unless its **checksum** is not valid

Stable Storage (2/2)



(a) (b) (c)

Recovery after a crash

- If the checksum of disk 1 is valid, and the two blocks are different, copy block from disk 1 to disk 2
- If the checksum of disk 1 is not valid, use the block on disk 2, if its checksum is valid
- If the checksums of both disks are not valid, then **data has been lost**, i.e. we have a **catastrophic-failure**

Further Reading

- Chapter 8, Tanenbaum and van Steen, *Distributed Systems*, 2nd Ed.
  - Section 8.5: Distributed Commit
  - Paragraph 8.6.1: Stable Storage
- P. Bernstein, V. Hadzilacos and N. Goodman, *Distributed Recovery*, Chp. 7 of *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987