Cours 7

Atomic Transaction & Distributed Atomic Transaction

- Example: online bank transaction:
  - withdraw(amount, account1)
  - deposit(amount, account2)
- What if network fails before deposit?
- Solution: Group operations in an atomic transaction.

The original model of the atomic transaction comes from the world of business

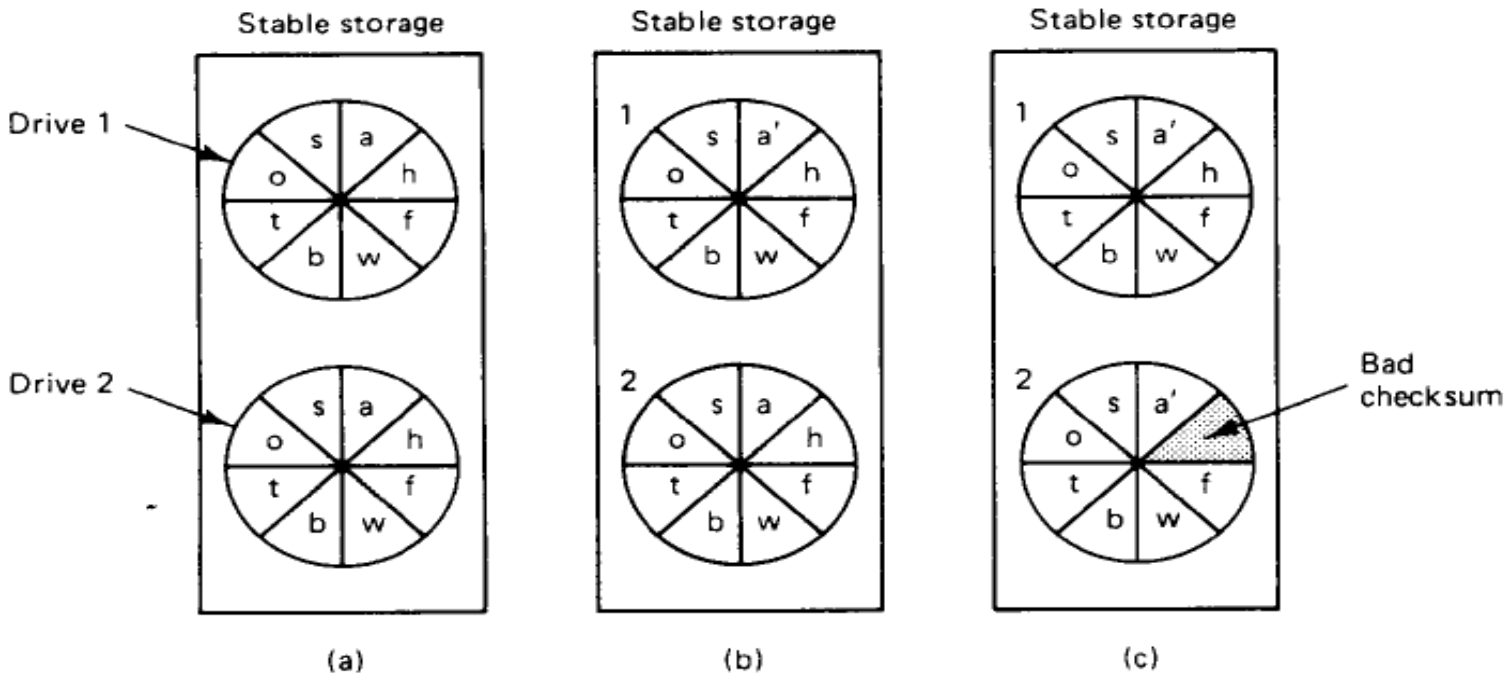


Fig. 3-15. (a) Stable storage. (b) Crash after drive 1 is updated. (c) Bad spot.

Volatile storage RAM && Disk Storage (DD)*vs.* *stable storage (Fig3-15)*

As a consequence of its implementation, stable storage is well suited to applications that require a high degree of fault tolerance, such as atomic transactions. When data are written to stable storage and then read back to check that they have been written correctly, the chance of them subsequently being lost is extremely small.

1. BEGIN_TRANSACTION: Mark the start of a transaction.
2. END_TRANSACTION: Terminate the transaction and try to commit.
3. ABORT_TRANSACTION: Kill the transaction; restore the old values.
4. READ: Read data from a file (or other object).
5. WRITE: Write data to a file (or other object).

Transaction primitives

- BEGIN_TRANSACTION
- reserve A-B
- reserve B-C
- reserve C-D
- END_TRANSACTION

BEGIN_TRANSACTION
reserve A-B
reserve B-C
reserve C-D **Complet**
=>ABORT_TRANSACTION

Example

1. Atomic: To the outside world, the transaction happens indivisibly.
2. Consistent: The transaction does not violate system invariants.
3. Isolated: Concurrent transactions do not interfere with each other.
4. Durable: Once a transaction commits, the changes are permanent.

ACID Properties

```
BEGIN TRANSACTION        BEGIN TRANSACTION        BEGIN TRANSACTION
  x := 0;                  x := 0;                  x := 0;
  x := x + 1;              x := x + 2;              x := x + 3;
END TRANSACTION          END TRANSACTION          END TRANSACTION
```

schedule 1	X=0	X=X+1	X=0	X=X+2	X=0	X=X+3	legal
schedule 2	X=0	X=0	X=X+1	X=X+2	X=0	X=X+3	legal
schedule 3	X=0	X=0	X=X+1	X=0	X=X+2	X=X+3	illegal

Isolation or Serializability

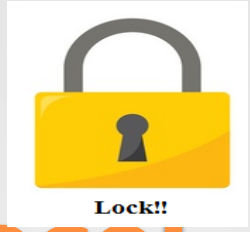
Concurrent Atomic Transactions

- If we allow the concurrent execution of mutiple transaction at the same time.
- The concurrent execution of transactions must be the same as if the transactions were executed serially in some arbitrary order. This property, called **serializability**
- **Solution :**
  - **Binary** semaphore '*mutex*'
  - When a transaction starts executing, its first action is to execute wait(*mutex).* After the transaction either commits or aborts, it executes signal(*mutex)*
  - it is too restrictive. As we shall see, in many cases we can allow transactions to overlap their execution while maintaining serializability.

Locking Protocol

- One way to ensure serializability is to associate a **lock** with each **data item** and to require that.
- Each transaction follow a **locking protocol that governs how** locks are acquired and released.

1. **Shared. If a transaction *Ti has obtained a shared-mode lock (denoted by*** S) on data item *Q, then Ti can read Q but cannot write it.*
2. **Exclusive. If a transaction *Ti has obtained an exclusive-mode lock (denoted*** by X) on data item *Q, then Ti can both read and write Q*



Locking Protocol

- Suppose that *Ti requests an exclusive lock on Q. In* this case, *Ti must wait until the lock on Q is released.*
- *If Ti requests a shared* lock on *Q, then Ti must wait if Q is locked in exclusive mode. Otherwise, it* can obtain the lock and access *Q.*
- Notice that this scheme is quite similar to a well known synchronized algorithm?

Two-phase locking protocol

- One protocol that ensures serializability.
- Each transaction issue lock and unlock requests in two phases:

1. **Growing phase.** A transaction may obtain locks but may not release any locks.

- **Shrinking phase.** A transaction may release locks but may not obtain any new locks.

Two-phase locking protocol

- Initially, a transaction is in the **growing phase**. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the **shrinking phase**, and no more lock requests can be issued.
- The two-phase locking protocol ensures *conflict-serializability*.

- **Le journal des inscriptions.**

The other common method of implementing transactions is the **writeahead log**, sometimes called an **intentions list**. With this method, files are actually modified in place, but before any block is changed, a record is written to the writeahead log on stable storage telling which transaction is making the change, which file and block is being changed, and what the old and new values are. Only after the log has been written successfully is the change made to the file.

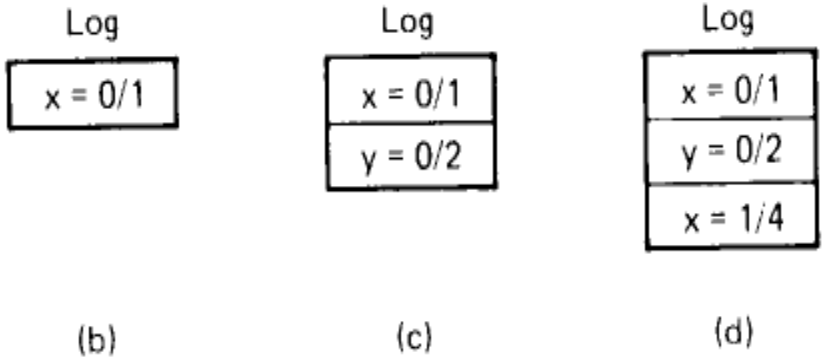
If the transaction succeeds and is committed, a commit record is written to the log, but the data structures do not have to be changed, as they have already been updated. If the transaction aborts, the log can be used to back up to the original state. Starting at the end and going backward, each log record is read and the change described in it undone. This action is called a **rollback**.

Implementation: comment implementer les transactions ? Example Writeahead Log

Figure 3-19 gives an example of how the log works. In Fig. 3-19(a) we have a simple transaction that uses two shared variables (or other objects), *x* and *y*, both initialized to 0. For each of the three statements inside the transaction, a log record is written before executing the statement, giving the old and new values, separated by a slash.

```
x = 0;
y = 0;
BEGIN_TRANSACTION
    x = x + 1;
    y = y + 2;
    x = y * y;
END_TRANSACTION
```

(a)



(b) (c) (d)

Fig. 3-19. (a) A transaction. (b)–(d) The log before each statement is executed.

Example Writeahead Log

Tools for Implementing Atomic Transactions
(Log File)

- *Begin_transaction*
  - Place a *begin* entry in log
- *Write*
  - Write updated data to log
- *Abort_transaction*
  - Place *abort* entry in log
- *End_transaction* (i.e., *commit*)
  - Place *commit* entry in log
  - *Copy logged data to files!*
  - Place *done* entry in log

Tools for Implementing Atomic Transactions (continued)

- Crash recovery – search log
  - If *begin* entry, look for matching entries :
  - If *done*, do nothing (all files have been updated)
  - If *abort*, undo any permanent changes that transaction may have made
  - If *commit* but not *done*, copy updated blocks from log to files, then add *done* entry

Distributed Commit

- When a transaction spans multiple sites, we require that either all sites commit the transaction or all abort it. This problem is called the ***distributed commit*** *problem*

Distributed Atomic Transactions

- Atomic transactions that span multiple sites and/or systems
- Same semantics as atomic transactions on single system
  - *A C I D*

General Solution – Two-phase Commit

- Textbook references
  - SILBERSCHATZ, GALVIN, GAGNE, Operating System Concepts with Java §18.3.1
  - Tanenbaum & Van Steen, §8.5
- Seminal reference
  - Jim Gray, "Notes on Database Operating Systems," in *Operating Systems: an Advanced Course*, vol 60 of *Lecture Notes in Comp. Sci.*, Springer-Verlag, 1978, pp. 393-481

Two-Phase Commit

- One site is elected *coordinator* of the transaction *T*
  - There is a whole theory of *election algorithms*
- *Phase 1:* When coordinator is ready to commit the transaction
  - Place *Prepare*(*T*) state in log on stable storage
  - Send *Vote_request*(*T*) message to all other participants
  - Wait for replies

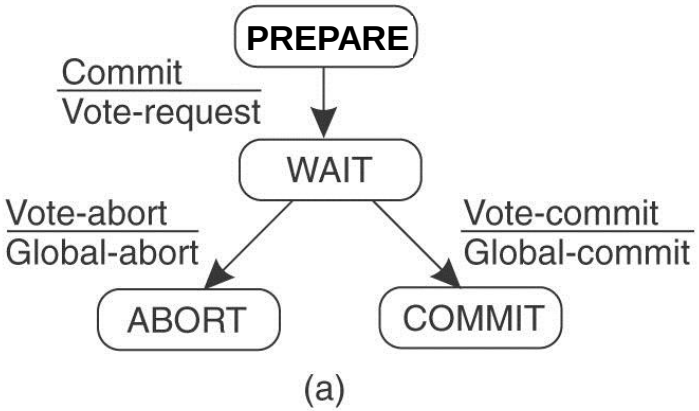
Two-Phase Commit (Coordinator – continued)

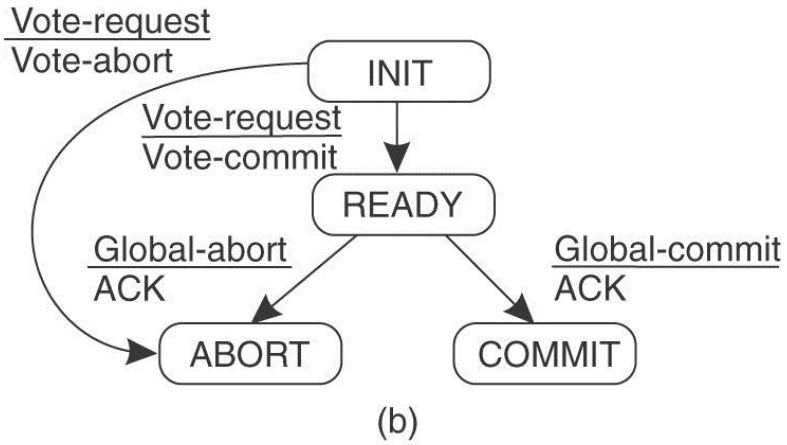
- *Phase 2:* Coordinator
  - If any participant replies *Abort*(*T*)
    - Place *Abort*(*T*) state in log on stable storage
    - Send *Global_Abort*(*T*) message to all participants
    - Locally abort transaction *T*
  - If *all* participants reply *Ready_to_commit*(*T*)
    - Place *Commit*(*T*) state in log on stable storage
    - Send *Global_Commit*(*T*) message to all participants
    - Proceed to commit transaction locally

Two-Phase Commit (Participant – continued)

- Phase I: Participant gets *Vote_request*(*T*) from coordinator
  - Place *Abort*(*T*) or *Ready*(T) state in local log
  - Reply with *Abort*(*T*) or *Ready_to_commit*(*T*) message to coordinator
  - If *Abort*(*T*) state, locally abort transaction
- Phase II: Participant
  - Wait for *Global_Abort*(*T*) or *Global_Commit*(*T*) message from coordinator
  - Place *Abort*(*T*) or *Commit*(T) state in local log
  - Abort or commit locally per message

Two-Phase Commit States





coordinator

participant