Principles of Software Construction:
Objects, Design, and Concurrency

Part 6: Concurrency and distributed systems

Transactions and Serializability

Jonathan Aldrich **Charlie Garrod**

School of
Computer Science

institute for
SOFTWARE
RESEARCH

# Administrivia

- Homework 6…
  - Checkpoint due tomorrow 5 p.m.
- Final exam Thurs Dec 17th, 8:30-11:30 am, MM 103 & MM A14
  - Final exam review session Wednesday, Dec 16th 2-4 p.m. DH 1112

# Key concepts from Tuesday

# Friends + friends of friends + friends of friends of friends

- E.g., for each person in a social network graph, count their friends and friends of friends and friends of friends of friends
  - For Map: `key1` is a person, `value` is the list of her friends
  - For Reduce: `key2` is ???, `values` is a list of ???

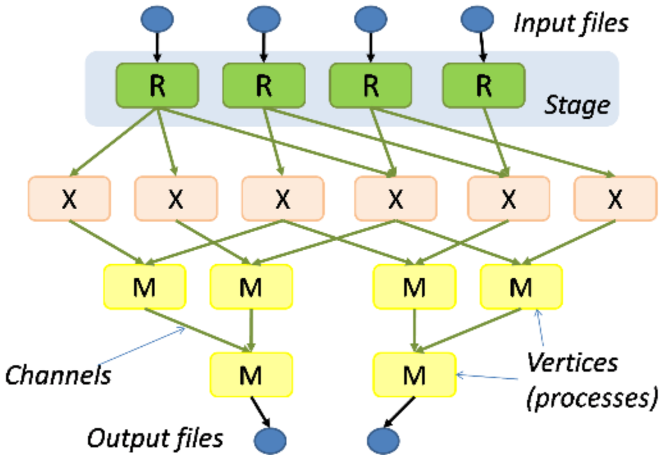
```
f1(String key1, String value):
```

```
f2(String key2, Iterator values):
```

MapReduce: (person, friends)* → (person, count of f + fof + fofof)*

# Dataflow processing

- High-level languages and systems for complex MapReduce-like processing
  - Yahoo Pig, Hive
  - Microsoft Dryad, Naiad
- MapReduce generalizations…



# Today:  Transactions and serializability

- A formal definition of consistency
- Introduction to transactions
- Concurrency control
- Distributed concurrency control
  - Two-phase commit

# An aside: Double-entry bookkeeping

- A style of accounting where every event consists of two separate entries: a credit and a debit

```
void transfer(Account fromAcct, Account toAcct, int val) {
    fromAccount.debit(val);
    toAccount.credit(val);
}

static final Account BANK_LIABILITIES = …;

void deposit(Account toAcct, int val) {
    transfer(BANK_LIABILITIES, toAcct, val);
}

boolean withdraw(Account fromAcct, int val) {
    if (fromAcct.getBalance() < val) return false;
    transfer(fromAcct, BANK_LIABILITIES, val);
    return true;
}
```

# Some properties of double-entry bookkeeping

- Redundancy!
- Sum of all accounts is static
  - Can be 0

# Data consistency of an application

- Suppose $\mathcal{D}$ is the database for some application and $\varphi$ is a function from database states to {true, false}
  - We call $\varphi$ an *integrity constraint* for the application if $\varphi(\mathcal{D})$ is true if the state $\mathcal{D}$ is "good"
  - We say a database state $\mathcal{D}$ is *consistent* if $\varphi(\mathcal{D})$ is true for all integrity constraints $\varphi$
  - We say $\mathcal{D}$ is inconsistent if $\varphi(\mathcal{D})$ is false for any integrity constraint $\varphi$

# Data consistency of an application

- Suppose $\mathcal{D}$ is the database for some application and $\varphi$ is a function from database states to {true, false}
  - We call $\varphi$ an *integrity constraint* for the application if $\varphi(\mathcal{D})$ is true if the state $\mathcal{D}$ is "good"
  - We say a database state $\mathcal{D}$ is *consistent* if $\varphi(\mathcal{D})$ is true for all integrity constraints $\varphi$
  - We say $\mathcal{D}$ is inconsistent if $\varphi(\mathcal{D})$ is false for any integrity constraint $\varphi$
- E.g., for a bank using double-entry bookkeeping one possible integrity constraint is:

```
def IsConsistent(D):
    If sum(all account balances in D) == 0:
        Return True
    Else:
        Return False
```

# Database transactions

- A *transaction* is an atomic sequence of read and write operations (along with any computational steps) that takes a database from one state to another
  - "*Atomic*" ~ indivisible
- Transactions always terminate with either:
  - *Commit*: complete transaction's changes successfully
  - *Abort*: undo any partial work of the transaction

# Database transactions

- A *transaction* is an atomic sequence of read and write operations (along with any computational steps) that takes a database from one state to another
  - "*Atomic*" ~ indivisible
- Transactions always terminate with either:
  - *Commit*: complete transaction's changes successfully
  - *Abort*: undo any partial work of the transaction

```
boolean withdraw(Account fromAcct, int val) {
    begin_transaction();
    if (fromAcct.getBalance() < val) {
        abort_transaction();
        return false;
    }
    transfer(fromAcct, BANK_LIABILITIES, val);
    commit_transaction();
    return true;
}
```

# A functional view of transactions

- A transaction $\mathcal{T}$ is a function that takes the database from one state $\mathcal{D}$ to another state $\mathcal{T}(\mathcal{D})$
- In a correct application, if $\mathcal{D}$ is consistent then $\mathcal{T}(\mathcal{D})$ is consistent for all transactions $\mathcal{T}$

# A functional view of transactions

- A transaction $\mathcal{T}$ is a function that takes the database from one state $\mathcal{D}$ to another state $\mathcal{T}(\mathcal{D})$
- In a correct application, if $\mathcal{D}$ is consistent then $\mathcal{T}(\mathcal{D})$ is consistent for all transactions $\mathcal{T}$
  - E.g., in a correct application any serial execution of multiple transactions takes the database from one consistent state to another consistent state

# Database transactions in practice

- The application requests commit or abort, but the database may arbitrarily abort any transaction
  - Application can restart an aborted transaction
- Transaction ACID properties:
  - Atomicity: All or nothing
  - Consistency: Application-dependent as before
  - Isolation: Each transaction runs as if alone
  - Durability: Database will not abort or undo work of a transaction after it confirms the commit

# Concurrent transactions and serializability

- For good performance, database interleaves operations of concurrent transactions

# Concurrent transactions and serializability

- For good performance, database interleaves operations of concurrent transactions
- Problems to avoid:
  - Lost updates
    - Another transaction overwrites your update, based on old data
  - Inconsistent retrievals
    - Reading partial writes by another transaction
    - Reading writes by another transaction that subsequently aborts
- A schedule of transaction operations is *serializable* if it is equivalent to some serial ordering of the transactions

# Concurrency control for a database

- Two-phase locking (2PL)
  - Phase 1: acquire locks
  - Phase 2: release locks
- E.g.,
  - Lock an object before reading or writing it
  - Don't release any locks until commit or abort

# Concurrency control for a distributed database

- Distributed two-phase locking
  - Phase 1: acquire locks
  - Phase 2: release locks
- E.g.,
  - Lock all copies of an object before reading or writing it
  - Don't release any locks until commit or abort
- Two new problems:
  - Distributed deadlocks are possible
  - All participants must agree on whether each transaction commits or aborts

# Two-phase commit (2PC)

- Two roles:
  - Coordinator:  for each transaction there is a unique server coordinating the 2PC protocol
  - Participants:  any server storing data locked by the transaction
- Two phases:
  - Phase 1:  Voting (or Prepare) phase
  - Phase 2:  Commit phase
- Failure model:
  - Unreliable network:
    - Messages may be delayed or lost
  - Unreliable servers with reliable storage:
    - Servers may fail, but will eventually recover persistently-stored state

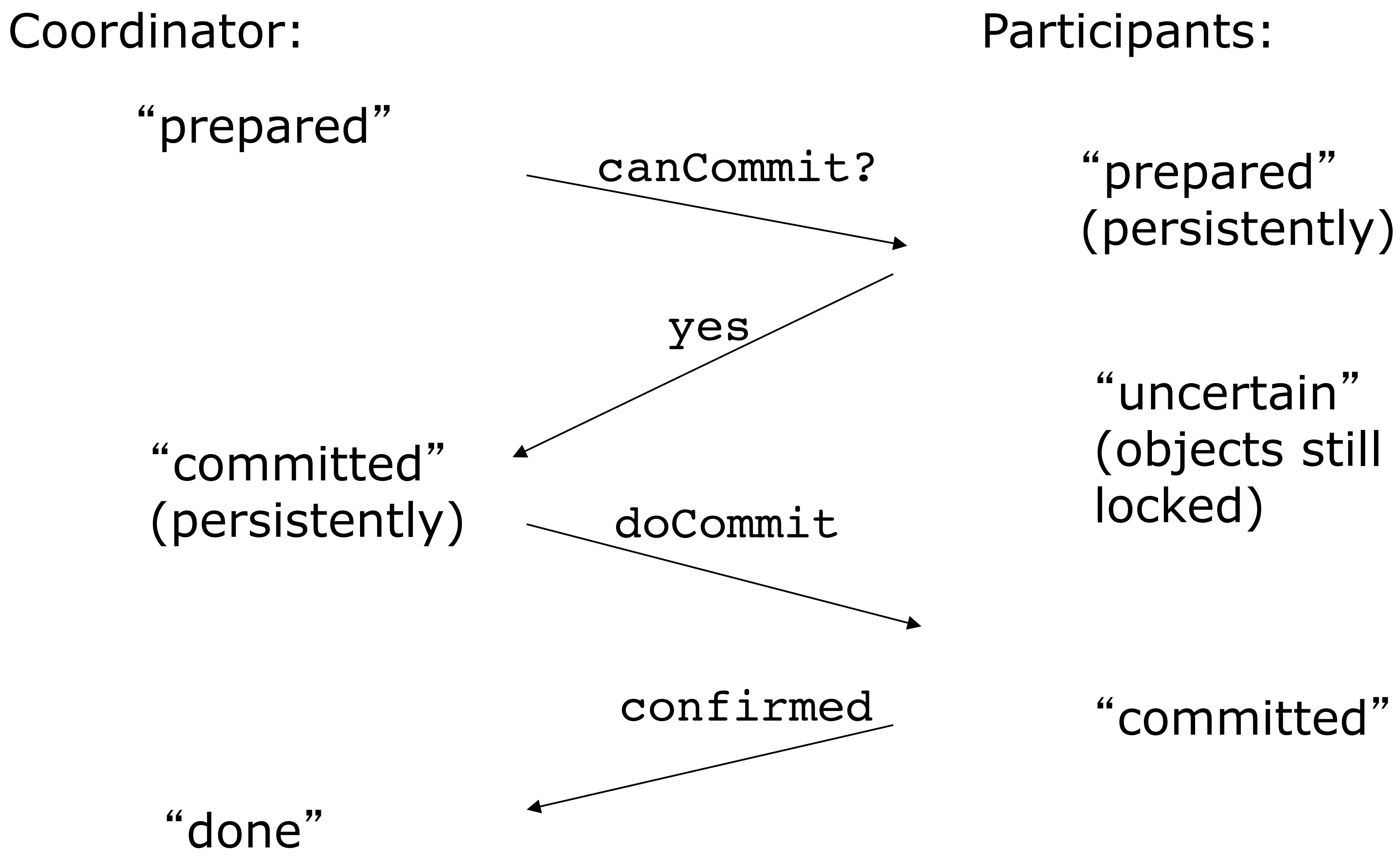
# The 2PC voting phase

- Coordinator sends `canCommit?`($\mathcal{T}$) message to each participant
  - Messages re-sent as needed
- Each participant replies `yes` or `no`
  - May not change vote after voting
    - Must log vote to persistent storage
    - If vote is `yes`:
      - Objects must be strictly locked to prevent new conflicts
      - Must log any information needed to successfully commit
- Coordinator collects replies from participants

# The 2PC commit phase

- If participants unanimously voted `yes`
  - Coordinator logs `commit(T)` message to persistent storage
  - Coordinator sends `doCommit(T)` message to all participants
    - Participants confirm, messages re-sent as needed
- If any participant votes `no`
  - Coordinator sends `doAbort(T)` message to all participants
    - Participants confirm, messages re-sent as needed

# 2PC sequence of events for a successful commit



# Problems with two-phase commit?

# Problems with two-phase commit?

- Failure assumptions are too strong
  - Real servers can fail permanently
  - Persistent storage can fail permanently
- Temporary failures can arbitrarily delay a commit
- Poor performance
  - Many round-trip messages

# The CAP theorem for distributed systems

- For any distributed system you want…
  - Consistency
  - Availability
  - tolerance of network Partitions
- …but you can support at most two of the three

# Next time…