Objects Analysis Threads Design 15-214

Principles of Software Construction: Objects, Design, and Concurrency

Distributed System Design, Part 4 MapReduce, continued, plus Transactions and Serializability

Fall 2014

Charlie Garrod Jonathan Aldrich

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

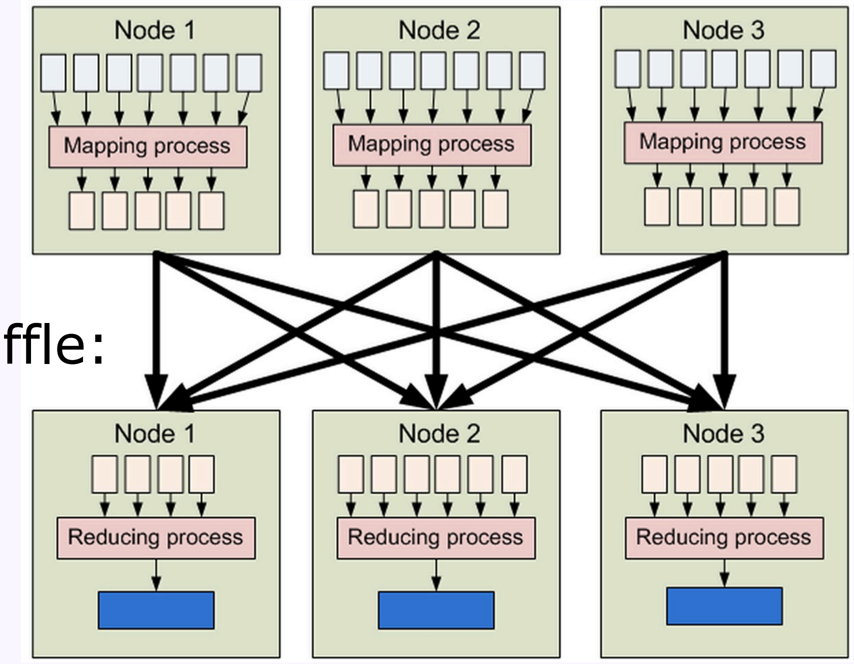
- Homework 5c due tonight
- Homework 6 available tomorrow morning
  - Checkpoint due Tuesday, December 2nd
  - Due Thursday, December 4th
  - Late days to Saturday, December 6th
- Final exam Monday, December 8th
  - Review session Sunday, Dec. 7th, noon – 3 p.m. DH 1212

Key concepts from Tuesday

MapReduce with key/value pairs (Google style)

- Master
  - Assign tasks to workers
  - Ping workers to test for failures
- Map workers
  - Map for each key/value pair
  - Emit intermediate key/value pairs

the shuffle:



- Reduce workers
  - Sort data by intermediate key and aggregate by key
  - Reduce for each key

MapReduce with key/value pairs (Google style)

- E.g., for each word on the Web, count the number of times that word occurs
  - For Map: `key1` is a document name, `value` is the contents of that document
  - For Reduce: `key2` is a word, `values` is a list of the number of counts of that word

```
f1(String key1, String value):
  for each word w in value:
    EmitIntermediate(w, 1);
```

```
f2(String key2, Iterator values):
  int result = 0;
  for each v in values:
    result += v;
  Emit(key2, result);
```

Map: (key1, v1) → (key2, v2)* Reduce: (key2, v2*) → (key3, v3)*

MapReduce: (key1, v1)* → (key3, v3)*

MapReduce: (docName, docText)* → (word, wordCount)*

Today: Distributed system design

- A few more MapReduce client problems
- Data consistency and concurrency control
  - A formal definition of consistency
  - Introduction to transactions
  - Serializability theory and concurrency control
  - Distributed concurrency control
    - Two-phase commit

MapReduce to count mutual friends

- E.g., for person in a social network graph, output the number of mutual friends they have
  - For Map: `key1` is a person, `value` is the list of her friends
  - For Reduce: `key2` is a pair of people, `values` is a list of 1s, for each mutual friend that pair has

```
f1(String key1, String value):
  for each pair of friends
         in value:
    EmitIntermediate(pair, 1);
```

```
f2(String key2, Iterator values):
  int result = 0;
  for each v in values:
    result += v;
  Emit(key2, result);
```

MapReduce: (person, friends)* → (pair of people, count of mutual friends)*

MapReduce to count incoming links

- E.g., for each page on the Web, count the number of pages that link to it
  - For Map: `key1` is a document name, `value` is the contents of that document
  - For Reduce: `key2` is ???, `values` is a list of ???

```
f1(String key1, String value):
  for each link in value:
    EmitIntermediate(link, 1)
```

```
f2(String key2, Iterator values):
  int result = 0;
  for each v in values:
    result += v;
  Emit(key2, result);
```

MapReduce: (docName, docText)* → (docName, number of incoming links)*

MapReduce to create an inverted index

- E.g., for each page on the Web, create a list of the pages that link to it
  - For Map: `key1` is a document name, `value` is the contents of that document
  - For Reduce: `key2` is ???, `values` is a list of ???

```
f1(String key1, String value):
  for each link in value:
    EmitIntermediate(link, key1)
```

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

MapReduce: (docName, docText)* → (docName, list of incoming links)*

List the mutual friends

- E.g., for each pair in a social network graph, list the mutual friends they have
  - 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)* → (pair of people, list of mutual friends)*

List the mutual friends

- E.g., for each pair in a social network graph, list the mutual friends they have
  - 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):

  for each pair of friends
        in value:

    EmitIntermediate(pair, key1);
```

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

  Emit(key2, values)
```

MapReduce: (person, friends)* → (pair of people, list of mutual friends)*

Count friends + friends of friends

- E.g., for each person in a social network graph, count their friends and 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)*

Count friends + friends of friends

- E.g., for each person in a social network graph, count their friends and 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):
  for each friend1 in value:
    EmitIntermediate(friend1, key1)
    for each friend2 in value:
      EmitIntermediate(friend1,
                       friend2);
```

```
f2(String key2, Iterator values):
  distinct_values = {}
  for each v in values:
    if not v in distinct_values:
      distinct_values.insert(v)
  Emit(key2, len(distinct_values))
```

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

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):
```

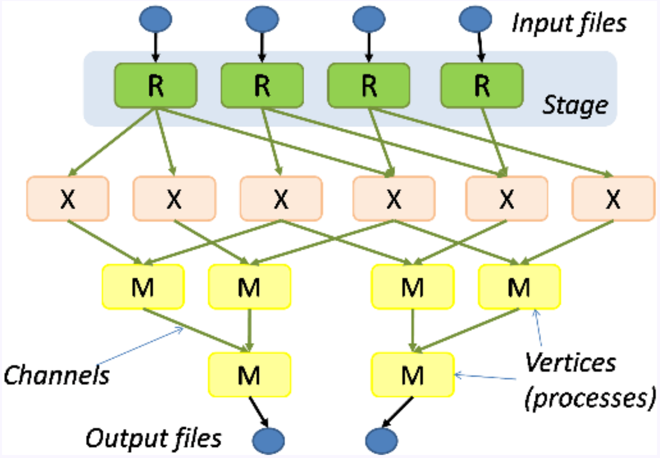
MapReduce: (person, friends)* → (person, count of f + fof + fofof)*

Problem: How to reach distance 3 nodes?

- Solution: Iterative MapReduce
  - Use MapReduce to get distance 1 and distance 2 nodes
  - Feed results as input to a second MapReduce process
- Also consider:
  - Breadth-first search
  - PageRank
  - …

Dataflow processing

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



Today: Distributed system design

- A few more MapReduce client problems
- Data consistency and concurrency control
  - A formal definition of consistency
  - Introduction to transactions
  - Serializability theory and 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 φ is a function from database states to {true, false}
  - We call φ 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 φ
  - We say $\mathcal{D}$ is inconsistent if $\varphi(\mathcal{D})$ is false for any integrity constraint φ

Data consistency of an application

- Suppose $\mathcal{D}$ is the database for some application and φ is a function from database states to {true, false}
  - We call φ 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 φ
  - We say $\mathcal{D}$ is inconsistent if $\varphi(\mathcal{D})$ is false for any integrity constraint φ

- 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
  - a.k.a. *linearizable*

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 crash or temporarily fail
    - Will eventually recover persistently-stored state

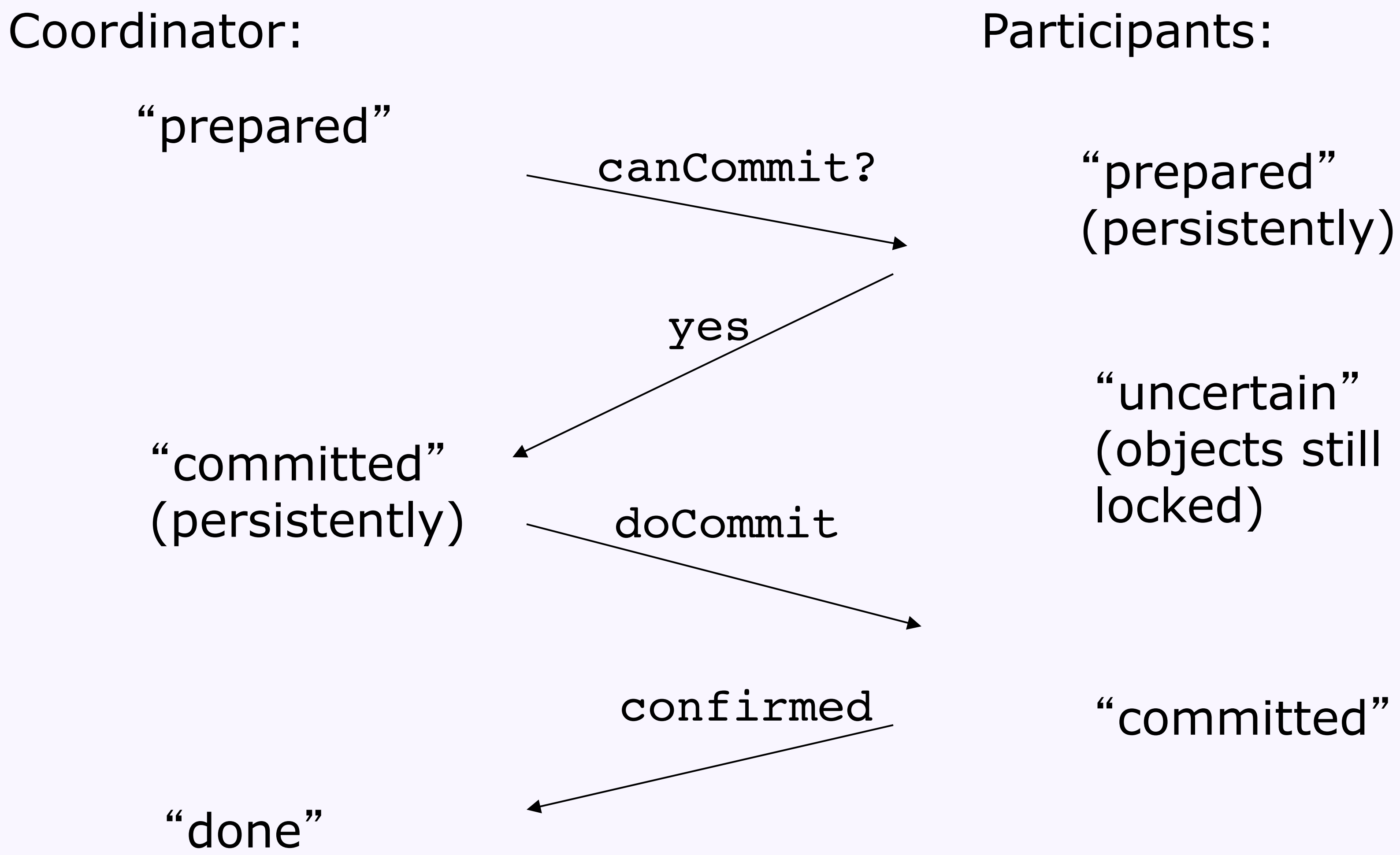
The 2PC voting phase

- 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 time sequence of events



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 week…