Transactions

CS 475, Spring 2018
Concurrent & Distributed Systems
Review: Strawman Sharding Scheme

In this class:

40 students

16 students
Review: Hashing

- Compresses data: maps a variable-length input to a fixed-length output
- Relatively easy to compute
- Example:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Hash Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leo McGarry</td>
<td>0</td>
</tr>
<tr>
<td>Josh Lyman</td>
<td>1</td>
</tr>
<tr>
<td>Sam Seaborn</td>
<td>2</td>
</tr>
<tr>
<td>Toby Ziegler</td>
<td>3</td>
</tr>
</tbody>
</table>

Hash

0
1
2
3
4
5
6
7
Review: Consistent Hashing

- Construction:
  - Assign each of C hash buckets to random points on mod $2^n$ circle, where hash key size $= n$
  - Map object to pseudo-random position on circle
  - Hash of object is the closest clockwise bucket

Example: hash key size is 16
Each \( \bullet \) is a value of hash $\%$ 16
Each \( \bullet \) is a bucket
Example: bucket with key 9?
Review: Consistent Hashing

It is relatively smooth: adding a new bucket doesn't change that much

Add new bucket: only changes location of keys 7,8,9,10

Delete bucket: only changes location of keys 1,2,3
Announcements

• HW4 is out!

• Today:
  • Beginning to talk about fault tolerance
  • Agreement & transactions in distributed systems

• Reminder, today:
  • Video lecture (Prof Bell at secret meeting)
Fault Tolerance

Set A=5

“OK”!

Read A

“5”!

Set A=5

“OK!”
boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount) {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}

Assume running on a single machine: What can go wrong here?
Transactions: Classic Example

boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount) {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}

What’s wrong here?
Need isolation (prevent overdrawing)
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount) {
    synchronized (from) {
        if (from.balance >= amount) {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
}
```

Adding a lock: prevents accounts from being overdrawn

```java
transferMoney(P1, P2, 100)
P1.balance (200) >= 100
P1.balance = 200 - 100 = 100
P2.balance = 200 + 100 = 300
return true;
```

```java
transferMoney(P1, P2, 200)
P1.balance <= 200
return false;
```

But: shouldn’t we lock on to also?
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount) {
    synchronized (from, to) {
        if (from.balance >= amount) {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
    }
    return false;
}
```

Locking on both from, to at same time

- `transferMoney(P1, P2, 100)`
  - P1.balance (200) >= 100
  - P1.balance = 200 - 100 = 0
  - P2.balance = 200 + 100 = 300
  - return true;

- `transferMoney(P1, P2, 200)`
  - P1.balance <= 200
  - return false;
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount) {
    synchronized (from, to) {
        if (from.balance >= amount) {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
}
```

```java
transferMoney(P1, P2, 100)
P1.balance (200) >= 100
P1.balance = 200 - 100 = 0

transferMoney(P1, P2, 200)
P1.balance <= 200
return false;
```

Problem: P1.balance was deducted P2.balance not incremented! ("Atomicity violation")
Transactions

• How can we provide some consistency guarantees across operations
• Transaction: unit of work (grouping) of operations
  • Begin transaction
  • Do stuff
  • Commit OR abort
Properties of Transactions

• Traditional properties: ACID
  • **Atomicity**: transactions are “all or nothing”
  • **Consistency**: Guarantee some basic properties of data; each transaction leaves the database in a valid state
  • **Isolation**: Each transaction runs as if it is the only one; there is some valid serial ordering that represents what happens when transactions run concurrently
  • **Durability**: Once committed, updates cannot be lost despite failures
Distributed Transactions

- What we are most interested in
- What happens if you need to coordinate e.g. transfers between multiple banks?

```plaintext
transferMoney("from": Barney@Goliath National, "to": Mortimer@ Duke&Duke, "amount"=1)
```

Initially: Barney.balance= $10000, Mortimer.balance= $10000
Distributed Transactions

• System model: data stored in multiple locations, multiple servers participating in a single transaction. One server pre-designated “coordinator”

• Failure model: messages can be delayed or lost, servers might crash, but have persistent storage to recover from
Distributed Transactions

- Coordinator: Begins a transaction
  - Assigns a unique transaction ID
  - Responsible for commit + abort
  - In principle, any client can be the coordinator, but all participants need to agree on who is the coordinator
- Participants: everyone else who has the data used in the transaction
1-Phase Commit (no transactions)

We couldn’t successfully commit on all 3 machines. But 1-phase commit has no way to go back!
1-Phase Non-Transaction Commit

• Naive protocol: coordinator broadcasts out “commit!” continuously until participants all say “OK!”

• Problem: what happens when a participants fails during commit? How do the other participants know that they shouldn’t have really committed and they need to abort?
2-Phase Commit

- Separate the commit into two steps:
  - 1: Voting
    - Each participant prepares to commit and votes of whether or not it can commit
  - 2: Committing
    - Once voting succeeds, every participant commits or aborts
2PC: Voting

- Coordinator asks each participant: can you commit for this transaction?
- Each participant prepares to commit BEFORE answering yes
  - e.g. save transaction to disk for later recovery
  - Can not abort after saying yes
- Outcome of transaction is unknown until the coordinator receives all votes and says “do abort” or “do commit”
2PC Event Sequence

Coordinator
Transaction state:
- prepared
- committed
- done

Participant
Local state:
- prepared
- uncertain
- committed

Can you commit?
Yes
OK, commit
OK I committed
2PC Example

```python
transferMoney("from": Barney@Goliath National, "to": Mortimer@ Duke&Duke, "amount"=$1)
```

Initially: Barney.balance= $10000, Mortimer.balance=$10000

Requirements:
1. Atomicity (transfer happens or doesn’t)
2. Concurrency control (serializability)
2PC Example

For simplicity, let’s assume transfer is:

```cpp
int transfer(src, dst, amt) {
    transaction = begin();
    src.bal -= amt;
    dst.bal += amt;
    return transaction.commit();
}
```
2PC Example

Coordinator (client or 3rd party) 

Goliath National

Duke & Duke

transaction.commit()

prepare

response_{GNB}

response_{D&D}

outcome

If we can commit, then lock our customer, vote “yes”

If everyone can commit, then outcome == commit, else abort

outcome

outcome
Fault Recovery

• How do we recover transaction state if we crash?
• Goal:
  • Committed transactions are not lost
  • Non-committed transactions either continue where they were or aborted
Recovery in 2PC

• What to log?
  • State changes in protocol
  • Participants: prepared; uncertain; committed/aborted
  • Coordinator: prepared; committed/aborted; done
  • These messages are idempotent - can be repeated

• Recovery depends on failure
  • Crash + reboot + recover
  • Timeout + recover
Crash + Reboot Recovery

- Nodes can’t back out once commit is decided
- If coordinator crashes just AFTER deciding “commit”
  - Must remember this decision, replay
- If participant crashes after saying “yes, commit”
  - Must remember this decision, replay
- Hence, all nodes need to log their progress in the protocol
2PC Example with logging

Coordinator (client or 3rd party)

Participant Goliath National

Participant Duke & Duke

transaction.commit()

Log!

prepare

response_{GNB}

Log!

prepare

Log!

Log!

response_{D&D}

If we can commit, then lock our customer, vote “yes”

outcome

Log!

outcome

Log!

If everyone can commit, then outcome == commit, else abort
Recovery on Reboot

• If coordinator finds no “commit” message on disk, abort
• If coordinator finds “commit” message, commit
• If participant finds no “yes, ok” message, abort
• If participant finds “yes, ok” message, then replay that message and continue protocol
Timeouts in 2PC

• Example:
  • Coordinator times out waiting for Goliath National Bank’s response
  • Bank times out waiting for coordinator’s outcome message

• Causes?
  • Network
  • Overloaded hosts
  • Both are very realistic…
Coordinator Timeouts

- If coordinator times out waiting to hear from a bank
  - Coordinator hasn’t sent any commit messages yet
  - Can safely abort - send abort message
  - Preserves correctness, sacrifices performance (maybe didn’t need to abort!)
- If either bank decided to commit, it’s fine - they will eventually abort
Handling Bank Timeouts

• What if the bank doesn’t hear back from coordinator?
• If bank voted “no”, it’s OK to abort
• If bank voted “yes”
  • It can’t decide to abort (maybe both banks voted “yes” and coordinator heard this)
  • It can’t decide to commit (maybe other bank voted yes)
• Does bank just wait for ever?
Handling Bank Timeouts

• Can resolve SOME timeout problems with guaranteed correctness in event bank voted “yes” to commit
• Bank asks other bank for status (if it heard from coordinator)
• If other bank heard “commit” or “abort” then do that
• If other bank didn’t hear
  • but other voted “no”: both banks abort
  • but other voted “yes”: no decision possible!
2PC Timeouts

• We can solve a lot (but not all of the cases) by having the participants talk to each other
• But, if coordinator fails, there are cases where everyone stalls until it recovers
• Can the coordinator fail?… yes
• We’ll come back to this “discuss amongst yourselves” kind of transactions next week