Logistics

• MP3 has been released!
Distributed Transactions and Replication

- Objects distributed among 1000’s cluster nodes for load-balancing (sharding)

- Objects replicated among a handful of nodes for availability / durability.
  - Replication across data centers, too

- Two-level operation:
  - Use transactions, coordinators, 2PC per object
  - Use Paxos / Raft among object replicas

- Consensus needed across object replicas, e.g.
  - When acquiring locks and executing operations
  - When committing transactions
2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs
2PC and Paxos

• E.g. workflow:
  • Coordinator leader sends Prepare message to leaders of each replica group
  • Each replica leader uses Paxos to commit the Prepare to the group logs
2PC and Paxos

• E.g. workflow:
  • Coordinator leader sends Prepare message to leaders of each replica group
  • Each replica leader uses Paxos to commit the Prepare to the group logs
2PC and Paxos

• E.g. workflow:
  • Coordinator leader sends Prepare message to leaders of each replica group
  • Each replica leader uses Paxos to commit the Prepare to the group logs
2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs
2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs
  - Once commit prepare succeeds, reply to coordinator leader
2PC and Paxos

E.g. workflow:
- Coordinator leader sends Prepare message to leaders of each replica group
- Each replica leader uses Paxos to commit the Prepare to the group logs
- Once commit prepare succeeds, reply to coordinator leader
- Coordinator leader uses Paxos to commit decision to its group log.
2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs
  - Once commit prepare succeeds, reply to coordinator leader
  - Coordinator leader uses Paxos to commit decision to its group log.
  - Coordinator leader sends Commit message to leaders of each replica group.
2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs
  - Once commit prepare succeeds, reply to coordinator leader
  - Coordinator leader uses Paxos to commit decision to its group log.
  - Coordinator leader sends Commit message to leaders of each replica group.
  - Each replica leader uses Paxos to process the final commit.
2PC and Paxos

• E.g. workflow:
  • Coordinator leader sends Prepare message to leaders of each replica group
  • Each replica leader uses Paxos to commit the Prepare to the group logs
  • Once commit prepare succeeds, reply to coordinator leader
  • Coordinator leader uses Paxos to commit decision to its group log.
  • Coordinator leader sends Commit message to leaders of each replica group.
  • Each replica leader uses Paxos to process the final commit.
  • Replica leader send the “commit ok / have committed” message back to coordinator.
Distributed Transactions and Replication

- Transaction processing can be distributed across multiple servers.
  - Different objects can be stored on different servers.
  - An object may be replicated across multiple servers.

- Case study: Google’s Spanner System
Spanner: Google’s Globally-Distributed Database

• First three lines from the paper:

  • Spanner is a scalable, globally-distributed database designed, built, and deployed at Google.

  • At the highest level of abstraction, it is a database that shards data across many sets of Paxos state machines in datacenters spread all over the world.

  • Replication is used for global availability and geographic locality; clients automatically fail over between replicas.
Spanner: Google’s Globally-Distributed Database

Wilson Hsieh
representing a host of authors
OSDI 2012
What is Spanner?

- Distributed multiversion database
  - General-purpose transactions (ACID)
  - SQL query language
  - Schematized tables
  - Semi-relational data model

- Running in production
  - Storage for Google’s ad data
  - Replaced a sharded MySQL database
Example: Social Network

- San Francisco
- Seattle
- Arizona
  - US

- Brazil
  - Sao Paulo
  - Santiago
  - Buenos Aires

- Spain
  - Madrid
  - Lisbon

- Russia
  - Moscow
  - Berlin
  - Krakow

User posts
Friend lists

19,000,000

OSDI 2012
Overview

• Feature: Lock-free distributed read transactions
• Property: External consistency of distributed transactions
  – First system at global scale
• Implementation: Integration of concurrency control, replication, and 2PC
  – Correctness and performance
• Enabling technology: TrueTime
  – Interval-based global time
Read Transactions

• Generate a page of friends’ recent posts
  – Consistent view of friend list and their posts

Why consistency matters
1. Remove untrustworthy person X as friend
2. Post P: “My government is repressive...”
Single Machine

Block writes

Friend1 post
Friend2 post
...
Friend999 post
Friend1000 post

User posts
Friend lists

Generate my page
Multiple Machines

Block writes

Friend1 post
Friend2 post
...
Friend999 post
Friend1000 post

User posts
Friend lists

User posts
Friend lists

Generate my page
Multiple Datacenters

Friend1 post
US

Friend2 post
Spain
...

Friend999 post
Brazil

Friend1000 post
Russia

Generate my page
Version Management

- Transactions that write use strict 2PL
  - Each transaction $T$ is assigned a timestamp $s$
  - Data written by $T$ is timestamped with $s$

<table>
<thead>
<tr>
<th>Time</th>
<th>&lt;8</th>
<th>8</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>My friends</td>
<td>[X]</td>
<td>[]</td>
<td></td>
</tr>
<tr>
<td>My posts</td>
<td></td>
<td></td>
<td>[P]</td>
</tr>
<tr>
<td>X’s friends</td>
<td>[me]</td>
<td>[]</td>
<td></td>
</tr>
</tbody>
</table>
Synchronizing Snapshots

Global wall-clock time

==

External Consistency:
Commit order respects global wall-time order

==

Timestamp order respects global wall-time order
given

timestamp order == commit order
Timestamps, Global Clock

• Strict two-phase locking for write transactions
• Assign timestamp while locks are held

OSDI 2012
Timestamp Invariants

- Timestamp order == commit order

- Timestamp order respects global wall-time order
TrueTime

- “Global wall-clock time” with bounded uncertainty

\[ \text{TT.now() - 2*\epsilon} \leq \text{time} \leq \text{TT.now() + 2*\epsilon} \]
Timestamps and TrueTime

Acquired locks

Pick $s = \text{TT.now().latest}$

Release locks

$s$ Wait until $\text{TT.now().earliest} > s$

Commit wait

average $\epsilon$
Commit Wait and Replication

- Acquired locks
- Start consensus
- Achieve consensus
- Notify slaves
- Release locks
- Pick s
- Commit wait done
Commit Wait and 2-Phase Commit

- Acquired locks
- Release locks
- Start logging
- Done logging
- Committed
- Notify participants of s
- Start logging
- Done logging
- Committed
- Notify participants of s
- Prepare Send s
- Compute s for each
- Commit wait done
- Compute overall s
- Compute overall s

OSDI 2012
Example

Time | <8 | 8 | 15
---|---|---|---
My friends | [X] | [] | 
My posts | | | [P]
X's friends | [me] | [] |
What Have We Covered?

• Lock-free read transactions across datacenters
• External consistency
• Timestamp assignment
• TrueTime
  – Uncertainty in time can be waited out
What Haven’t We Covered?

• How to read at the present time
• Atomic schema changes
  – Mostly non-blocking
  – Commit in the future
• Non-blocking reads in the past
  – At any sufficiently up-to-date replica
TrueTime Architecture

Compute reference [earliest, latest] = now ± ε
TrueTime implementation

\[ \text{now} = \text{reference now} + \text{local-clock offset} \]
\[ \varepsilon = \text{reference } \varepsilon + \text{worst-case local-clock drift} \]
What If a Clock Goes Rogue?

- Timestamp assignment would violate external consistency
- Empirically unlikely based on 1 year of data
  - Bad CPUs 6 times more likely than bad clocks
Network-Induced Uncertainty

Date
Mar 29  Mar 30  Mar 31  Apr 1
6AM  8AM  10AM  12PM

Date (April 13)
8AM  10AM  12PM

Epsilon (ms)

99.9
99
90

OSDI 2012
What’s in the Literature

- External consistency/linearizability
- Distributed databases
- Concurrency control
- Replication
- Time (NTP, Marzullo)
Future Work

• Improving TrueTime
  – Lower $\varepsilon < 1$ ms

• Building out database features
  – Finish implementing basic features
  – Efficiently support rich query patterns
Conclusions

• Reify clock uncertainty in time APIs
  – Known unknowns are better than unknown unknowns
  – Rethink algorithms to make use of uncertainty

• Stronger semantics are achievable
  – Greater scale != weaker semantics
Thanks

• To the Spanner team and customers
• To our shepherd and reviewers
• To lots of Googlers for feedback
• To you for listening!

• Questions?
MP3: Distributed Transactions

- [https://courses.grainger.illinois.edu/ece428/sp2023/mps/mp3.html](https://courses.grainger.illinois.edu/ece428/sp2023/mps/mp3.html)
- Lead TA: Sarthak Moorjani

**Task:**
- Build a distributed transaction system that satisfies ACI properties (you do not need to handle Durability).

**Objective:**
- Think through and implement algorithms for achieving atomicity and consistency with distributed transactions (two-phase commit), concurrency control (two-phase locking / timestamped ordering), deadlock detection.
MP3: Distributed Transactions

Use this information to establish communication across servers.
MP3: Distributed Transactions

`branch_name` | `config_file` | server A
---|---|---
`branch_name` | `config_file` | server B
`branch_name` | `config_file` | server C
`branch_name` | `config_file` | server D
`branch_name` | `config_file` | server E

sample `config_file`

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>sp23-CS425-0101.cs.illinois.edu</td>
<td>1234</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>sp23-CS425-0101.cs.illinois.edu</td>
<td>1234</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>sp23-CS425-0101.cs.illinois.edu</td>
<td>1234</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>sp23-CS425-0101.cs.illinois.edu</td>
<td>1234</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>sp23-CS425-0101.cs.illinois.edu</td>
<td>1234</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`client_id` | `config_file`
MP3: Distributed Transactions

server A  server B  server C  server D  server E

client

Receives user input (command) from stdin.
Prints output of the command to stdout.

< BEGIN //start a new transaction
MP3: Distributed Transactions

- Server A
- Server B
- Server C
- Server D
- Server E

Client

Receives user input (command) from stdin. Prints output of the command to stdout.

- `<BEGIN //start a new transaction`
- `> OK`
- `<DEPOSIT A.foo 10 //deposit 10 units in account foo at branch A`

For each transaction, client randomly chooses a server to act as coordinator. Only communicates with the coordinator.
Receives user input (command) from stdin.
Prints output of the command to stdout.

< BEGIN //start a new transaction
> OK
< DEPOSIT A.foo 10 //deposit 10 units in account foo at branch A
> OK
MP3: Distributed Transactions

- server A
- server B
- server C
- server D
- server E

client

Receives user input (command) from stdin.
Prints output of the command to stdout.

< BEGIN //start a new transaction
> OK
< DEPOSIT A.foo 10 //deposit 10 units in account foo at branch A
> OK

Other possible commands: WITHDRAW and BALANCE (only applicable if the account exists)
User enters COMMIT or ABORT to end the transaction.

A server may also choose to ABORT a transaction (e.g. if consistency violated, or if needed for concurrency control).

Changes made by one transaction visible to others only after it successful commits.
Receives user input (command) from stdin.
Prints output of the command to stdout.

Required properties:
• Atomicity:
  • all servers commit the entire transaction, or all rollback the entire transaction.
• Consistency:
  • cannot withdraw from or read balance of a non-existent account.
  • a transaction cannot result in a negative account balance.
Receives user input (command) from stdin.
Prints output of the command to stdout.

Required properties:
• Isolation:
  • multiple clients may concurrently issue commands on the object.
  • Must provide serial equivalence.
• Deadlock avoidance.
MP3: Distributed Transactions

• Due on April 26th.
  • Late policy: Can use remainder of your 168 hours of grace period accounted per student over the entire semester.

• Read the specification fully and carefully.
  • Required semantics discussed more completely there.

• Start early!