Distributed Data Management

Christoph Lofi
Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de
For big web applications, we are often interested in features which are hard to get with RDBMS

- **Linear scalability and elasticity**
  - We want to add new machine with little overhead, performance should increase with each addition

- **Full availability**
  - The DB should never fail, and it should always accept read and writes
  - The DB should be disaster-proof

- **Flexibility during development and support for prototyping**
  - The Db should not enforce hard design time decisions, but should allow for changing the data model during runtime
These new requirements are often conflicting with typical RDBMS behavior and features

- We need to find suitable trade-offs

Example: Amazon Dynamo

- Focus on availability, scalability, and latency
- But:
  - No guaranteed consistency between replica
    - Asynchronous replication with eventual consistency
  - Simple key-value store
    - No complex data model, no query language
  - No traditional transaction support
Example: **Google BigTable**

- Still focus on availability and scalability, but introduce a slightly more powerful data model
  - One big “table” with multiple attributes

- But:
  - Still no guaranteed consistency between replica
    - Replication is still asynchronous
    - Data mutations and control flow separated to decrease conflict time
    - Append-only write model decrease conflict potential
  - Still, no complex query languages
  - Also, no traditional transaction support
Clearly, we see that one of the core problems is **replica consistency** and **transaction support** – What is the problem with that anyway?
• Three common approaches towards replica consistency:

  – **Synchronous Master-Slave Replication**
    • One master server holds the primary copy
    • Each mutation is pushed to all slaves (replicas)
    • Mutation is only acknowledged after each slave was mutated successfully
      – e.g., 2-Phase Commit Protocol
    • Advantage:
      – Replicas always consistent, even during failure
      – Very suitable for developing proper transaction protocols
    • Disadvantage:
      – Potentially unable to deal with partitions
      – System can be unavailable during failure states, bad scalability?
      – Failing master requires expensive repairs
Asynchronous Master-Slave

- e.g., Google BigTable
- Master writes mutations to at least one slave
  - Usually using write-ahead logs
  - Acknowledge write when log write is acknowledged by slave
  - For improved consistency, wait for more log acknowledgements
- Propagate mutation to other slaves as soon as possible
- Advantages:
  - Could be used for efficient transactions with some problems (see below)
- Disadvantages:
  - Could result in data loss or inconsistencies
    » Loss of master or first slave
    » Read during mutation propagation
  - Repair when loosing master can be complex
    » Need consensus protocol
– Optimistic Replication

• e.g., Amazon Dynamo
• All nodes are homogenous, no masters and no slaves
• Anybody can accept a mutations which are asynchronously pushed to replicas
• Acknowledgements after the first $n$ writes ($n$ may be 1)
• Advantages:
  – Very available
  – Very low latency
• Disadvantages:
  – Consistency problems can be quite common
  – No transaction protocols can be build on top because global mutation order is unknown
Core problem: When using asynchronous replication, and the system crashes, which state is correct?

- In Master-Slave setting: The master is always correct!
  - What happens if the master fails?
  - Expensive master failover recovery necessary

- In homogenous setting: Well... tough decision
  - We would need a perfect system for determining the global order of mutations
    - Vector clocks not good enough
    - Normal timestamps via internal clocks too unreliable
  - Maybe rely on consensus?
12.1 Paxos

• The PAXOS Problem:
  – How to reach **consensus/data consistency** in distributed system that can tolerate **non-malicious failures**?
  – Paxos made simple:
12.1 Paxos

• Paxos, a family of protocols for solving consensus in a network of unreliable processors
  – Unreliable but not malicious!

• Consensus protocols are the basis for the state machine replication approach
Why should we care about PAXOS?

- It can only find a consensus on a simple single value…
  - This is rather….useless?

- BUT: We can replicate transaction logs to multiple replica nodes, and then have PAXOS to find a consensus on log entries!
  - This can solve a distributed log problem
    - i.e. determining a serializable order of operations
  - Allows us to do a proper log-based transaction and recovery algorithms
Problem: **Find a consensus for the choice of the value of a data item**

- **Safety**
  - Only a value that has been proposed by some node may be chosen
  - Only a single value is chosen
  - Only chosen values are notified to nodes
    - Nodes learn about the value

- **Liveness**
  - Some proposed value is eventually chosen
  - If a value has been chosen, each node is eventually learns about the choice
12.1 Paxos notation

- **Classes of agents:**
  - **Proposers**
    - Takes a client request, and starts a voting phase, acting coordinator of the voting phase
  - **Acceptors** (or **Voters**)
    - Acceptors vote on the value of an item
    - Usually organized in quorums
      - i.e. a majority group of acceptors which can reliably vote on an item
      - Subset of acceptors
      - Any two quorums share at least one member
  - **Learners**
    - Learners replicate some functionality in the PAXOS protocol for reliability
      - e.g., communicate with client, repeating vote results, etc.

- **A node can act as more than one agent**
12.1 Paxos algorithm
12.1 Paxos algorithm

• Phase 1 (prepare):
  – A proposer (acting as the leader) creates a new proposal numbered \( n \) (increasing numbers)
    • Send prepare request to all acceptors
  – Acceptor receives a prepare request with number \( n \)
    • If \( n \) is greater than any previous prepare request, it accepts it and promises not to accept any lower numbered requests
      – Called a promise
      – If the acceptor ever accepted a lower numbered request in the past, return proposal number and value to proposer
    • If \( n \) is smaller than any previous prepare request, decline
12.1 Paxos algorithm

• Phase 2 (accept):
  – **Proposer**
    • If proposer receives enough promises from acceptors, it sets a **value** for its proposal
      – Enough means from a majority of acceptors (quorum)
      – If any acceptor had accepted a proposal before, the proposer got the respective value in the prepare phase
        » Use the value of the highest returned proposal
        » If no proposals returned by acceptors, choose a value
    • Send **accept message** to all acceptors in quorum with chosen value
  – **Accept**
    • Acceptor accepts proposal iff it has not promised to only accept higher proposals
      – i.e. accept if not already promised to vote on a newer proposal
      – Register the value, and send **accept message** to proposer and all learners
12.1 Definition of chosen

• A value is chosen at proposal number $n$ iff majority of acceptors accept that value in phase 2
  – As reported by learners

• No choice if (restart with higher proposal number)
  – When multiple proposers send conflicting prepare messages
  – When there is no quorum of responses
  – Eventually, there will be a consensus
12.1 Paxos properties

- P1: Any proposal number is unique
- P2: Any two set of acceptors (quorums) have at least one acceptor in common
- P3: The value sent out in phase 2 is the value of the highest-numbered proposal of all the responses in phase 1
12.1 Paxos by example

- **Proposer A**
  - Prepare request: $[n = 2, v=8]$

- **Proposer B**
  - Prepare request: $[n = 4, v=5]$

- **Acceptor X**
  - $[n = 2, v=8]$

- **Acceptor Y**
  - $[n = 2, v=8]$

- **Acceptor Z**
  - $[n = 4, v=5]$

[Sample from: http://angus.nyc/2012/paxos-by-example]
12.1 Paxos by example

- Proposer A
  - Proposer B
  - Acceptor X
    - $[n = 2, v=8]$
  - Acceptor Y
    - $[n = 2, v=8]$
  - Acceptor Z
    - $[n = 4, v=5]$

prepare response [no previous]
12.1 Paxos by example

- Acceptor Z ignores request from A because it already received higher request from B
- Proposer A then sends accept requests with n=2 and value v=8 to acceptors X and Y
  - These are then ignored as they already promised a proposal with n=4
12.1 Paxos by example

Proposer A

Proposer B

Acceptor X

Acceptor Y

Acceptor Z

Learner

- Proposer B sends accept request with highest received previous value v=8 to its quorum
12.1 Paxos by example

Proposer A

Proposer B

Acceptor X

Acceptor Y

Acceptor Z

Learner

- If an acceptor receives an accept request with higher or equal number that its highest seen proposal, it sends its value to each learner.
- A value is chosen when a learner get messages from a majority of its acceptors.
12.1 Proof of safety

• Claim: if a value $v$ is chosen at proposal number $n$, any value that is sent out in phase 2 of any later proposal numbers must be also $v$.

• Proof (by contradiction): Let $m$ is the first proposal number that is later than $n$ and in phase 2, the value sent out is not $v$. 
12.1 Learning a chosen value

• There are some options:
  – Each acceptor, whenever it accepts a proposal, informs all the learners
  – Acceptors informs a distinguished learner (usually the proposer) and let the distinguished learner broadcast the result
12.1 Tunable knobs

• Acceptors have many options to response:
  – Prepare request: No/Yes
  – Accept request: No/Yes if it didn’t promise not to do so
• Back off time after abandon a proposal: exponential back-off/pre-assigned values
• Should we wait for nodes to online in each phase?
12.1 Early applications

• Chubby lock service.

• Petal: Distributed virtual disks.

• Frangipani: A scalable distributed file system.
12.1 Paxos

• Often, PAXOS is considered to not scale too well
  – People believed that its not adaptable to Cloud applications
  – But…can we still manage somehow?

• Google Megastore!
  – Using PAXOS for building consistent scalable data stores ACID transactions!
12.2 Megastore

• **Megastore** was designed to run Google AppEngine
  – AppEngine developers demand more traditional DB features
    • Support for (simple) **schemas**
    • Support for secondary **indexes**
    • Support for (simple) **joins**
    • Support for (simple) **transactions**
  – Still, use Google infrastructure
    • Build on top of BigTable and GFS
    • Still have some scalability
• Each schema has a set of tables
  – Each table has a set of entities
  – Each entity has a set of strongly types properties
    • Can be required, optional, or repeated (multi-value)
    • So, no 1st normal form…
  – Each table needs a (usually composite) primary key
• Each table is either a entity group root table or a child table
  – There is a single foreign key between children and their root
  – An entity group is a root entity with all its child entities
12.2 Megastore - Schema

- Example schema:

```sql
CREATE SCHEMA PhotoApp;

CREATE TABLE User {
    required int64 user_id;
    required string name;
} PRIMARY KEY(user_id), ENTITY GROUP ROOT;

CREATE TABLE Photo {
    required int64 user_id;
    required int32 photo_id;
    required int64 time;
    required string full_url;
    optional string thumbnail_url;
    repeated string tag;
} PRIMARY KEY(user_id, photo_id),
IN TABLE User,
ENTITY GROUP KEY(user_id) REFERENCES User;

CREATE LOCAL INDEX PhotosByTime
ON Photo(user_id, time);

CREATE GLOBAL INDEX PhotosByTag
ON Photo(tag) STORING (thumbnail_url);
```
12.2 Megastore - Schema

- **Map Megastore schemas to Bigtable**
  - Each entity is mapped to a single Bigtable row
    - Use primary keys to cluster entities which will likely be read and accessed together
    - Entity groups will be stored consecutively
  - “IN TABLE” command forces an entity into a specific Bigtable
    - Bigtable column name = Megastore table name + property name
    - In this example: photos are stored close to users

<table>
<thead>
<tr>
<th>Row key</th>
<th>User. name</th>
<th>Photo. time</th>
<th>Photo. tag</th>
<th>Photo. _url</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>John</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101,500</td>
<td></td>
<td>12:30:01</td>
<td>Dinner, Paris</td>
<td></td>
</tr>
<tr>
<td>101,502</td>
<td></td>
<td>12:15:22</td>
<td>Betty, Paris</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>Mary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
You can have two levels of indexes

- **Local indexes** with each entity group
  - Stored in the group, and updated consistently and atomically

- **Global index**
  - Spans all entity groups of a table
  - Not necessarily consistent
  - Consistent update of global index
  - Requires additional latching (index locking)
    - Expensive
    - Many Web applications don’t need this anyway…
• Considerations:
  – We want **geo-replication** across multiple datacenters
    • But also replication within a datacenter
  – Data can probably be **sharded by user location**
    • Most read and writes for a data item will go to one specific datacenter
      – “**Master datacenter**”
    • Within one data center, there should be no master
      – Read and writes can be issued to any replica node
      – Better performance with respect to availability
• Simple assumption: Each entity group is a “Mini-Database”
  – Have *serializable ACID semantics* within each entity group
    • Serializable ACID >> Strict Consistency >> Eventual Consistency
  – Megastore uses the **MVCC protocol**
    • “MultiVersion Concurrency Control” with transaction timestamps
      – Timestamping can be used because transactions are handled within a single “master” datacenter
  – Isolate reads and writes
12.2 Megastore - Transactions

- What are MVCC transactions (using timestamps)?
12.3 Spanner

• Outline and Key Features
• System Architecture:
  – Software Stack
  – Directories
  – Data Model
  – TrueTime
• Evaluation
• Case Study

12.3 Motivation: Social Network

- US
  - San Francisco
  - Seattle
  - Arizona
- Brazil
  - Sao Paulo
  - Santiago
  - Buenos Aires
- Russia
  - Moscow
  - Berlin
  - Krakow

- Spain
  - London
  - Paris
  - Madrid
  - Lisbon

- Friend lists
  - User posts

- Growth: x1000
12.3 Outline

• Next step from Bigtable in RDBMS path with strong time semantics

• Key Features:
  – Temporal Multi-version database
  – Externally consistent global write-transactions with synchronous replication.
  – Transactions across Datacenters.
  – Lock-free read-only transactions.
  – Schematized, semi-relational (tabular) data model.
  – SQL-like query interface.
12.3 Key Features cont.

- Auto-sharding, auto-rebalancing, automatic failure response.
- Exposes control of data replication and placement to user/application.
- Enables transaction serialization via global timestamps.
- Acknowledges clock uncertainty and guarantees a bound on it.
- Uses novel TrueTime API to accomplish concurrency control.
- Uses GPS devices and Atomic clocks to get accurate time.
12.3 Server configuration

Universe: Spanner deployment
Zones: analogues to deployment of BigTable servers (units of physical isolation)
12.3 Spannserver Software Stack

- Participant leader
  - Transaction manager
  - Lock table

- Leader

- Replica
  - Paxos
  - Tablet
  - Colossus

- Data Center X
- Data Center Y
- Data Center Z
(key: string, timestamp: int64) → string

- Back End: Colossus (successor of GFS)
- To support replication:
  - each spanserver implements a **Paxos State Machine** on top of each tablet, and the state machine stores meta data and logs of its tablet.
- Set of replicas is collectively a **Paxos group**
• **Leader** among replicas in a Paxos group is chosen and all write requests for replicas in that group initiate at leader.

• At every replica that is a leader each spanserver implements:
  – a **lock table** and
  – a **transaction manager**
12.3 Directories

- Directory – analogous to bucket in BigTable
  - Smallest unit of data placement
  - Smallest unit to define replication properties
- Directory might in turn be sharded into Fragments if it grows too large.
12.3 Data model

- Query language expanded from SQL.
- Multi-version database: uses a version when storing data in a column (time stamp).
- Supports transactions and provides strong consistency.
- Database can contain unlimited schematized tables
• Not purely relational:
  – Requires rows to have names
  – Names are nothing but a set (can be singleton) of primary keys
  – In a way, it’s a key value store with primary keys mapped to non-key columns as values
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
INTERLEAVE IN PARENT Users ON DELETE CASCADE;

Implications of Interleave: hierarchy
12.3 TrueTime

- Novel API behind Spanner’s core innovation
- Leverages hardware features like GPS and Atomic Clocks
- Implemented via TrueTime API.

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>TT.now()</td>
<td>TTinterval: [earliest, latest]</td>
</tr>
<tr>
<td>TT.after(t)</td>
<td>True if t has passed</td>
</tr>
<tr>
<td>TT.before(t)</td>
<td>True if t has not arrived</td>
</tr>
</tbody>
</table>
• “Global wall-clock time” with bounded uncertainty
12.3 TrueTime implementation

- Set of time master server per datacenters and time slave daemon per machines.
- Majority of time masters are GPS fitted and few others are atomic clock fitted (Armageddon masters).
- Daemon polls variety of masters and reaches a consensus about correct timestamp.
12.3 TrueTime implementation

Diagram showing components of TrueTime implementation:
- Daemon
- Atomic oscillator time master
- GPS time master
12.3 TrueTime implementation

![Diagram showing the relationship between Time Daemon and Time Master with
the formula time = [t, t + e].]
12.3 TrueTime Architecture

Compute reference \([\text{earliest, latest}] = \text{now} \pm \varepsilon\)
12.3 TrueTime

• TrueTime uses both GPS and Atomic clocks since they are different failure rates and scenarios.

• Two other boolean methods in API are
  – After(t) – returns TRUE if t is definitely passed
  – Before(t) – returns TRUE if t is definitely not arrived

• TrueTime uses these methods in concurrency control and to serialize transactions.
12.3 TrueTime

• **After()** is used for Paxos Leader Leases
  – Uses after(Smax) to check if Smax is passed so that Paxos Leader can abdicate its slaves.

• **Paxos Leaders** can not assign timestamps(Si) greater than Smax for transactions(Ti) and clients can not see the data committed by transaction Ti till after(Si) is true.
  – After(t) – returns TRUE if t is definitely passed
  – Before(t) – returns TRUE if t is definitely not arrived

• **Replicas** maintain a timestamp tsafe which is the maximum timestamp at which that replica is up to date.
12.3 Concurrency control

1. Read-Write – requires lock.
2. Read-Only – lock free.
   - Requires declaration before start of transaction.
   - Reads information that is up to date
3. Snapshot Read – Read information from past by specifying a timestamp or bound
   - User specifies specific timestamp from past or timestamp bound so that data till that point will be read.
12.3 Timestamps

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

\[ T \]

Pick \( s = \text{now}() \)
12.3 Timestamp Invariants

- Timestamp order == commit order

- Timestamp order respects global wall-time order
12.3 Timestamps and TrueTime

- Acquired locks:
  - Pick \( s = \text{TT.now().latest} \)

- Release locks:
  - \( s \)
  - Wait until \( \text{TT.now().earliest} > s \)

Commit wait

- average \( \varepsilon \)
  - average \( \varepsilon \)
12.3 Commit Wait and Replication

Acquired locks

Start consensus
Achieve consensus
Notify slaves

Release locks

Pick s

Commit wait done
12.3 Commit Wait and 2-Phase Commit

**Diagram:**
- **T_C:**
  - Acquired locks
  - Start logging
  - Done logging
  - Release locks
  - Committed
  - Notify participants of s
- **T_P1:**
  - Acquired locks
  - Prepared
  - Send s
  - Compute overall s
- **T_P2:**
  - Acquired locks
  - Compute s for each
  - Send s
  - Commit wait done

**Steps:**
1. Acquired locks
2. Start logging
3. Done logging
4. Release locks
5. Committed
6. Notify participants of s
7. Prepared
8. Send s
9. Compute overall s
10. Compute s for each
11. Send s
12. Commit wait done
12.3 Example

- Remove X from my friend list
- Remove myself from X's friend list
- Risky post P

<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>[me]</td>
<td>[]</td>
<td>[P]</td>
</tr>
<tr>
<td>X's friends</td>
<td></td>
<td>[]</td>
<td></td>
</tr>
</tbody>
</table>
12.3 Evaluation

- Evaluated for replication, transactions and availability.
- Results on epsilon of TrueTime
- Benchmarked on Spanner System with
  - 50 Paxos groups
  - 250 Directories
  - Clients(applications) and Zones are at a network distance of 1ms
12.3 Evaluation - Availability

![Graph showing cumulative reads completed over time for different scenarios: non-leader, leader-soft, and leader-hard. The x-axis represents time in seconds, and the y-axis represents the cumulative number of reads completed in thousands (K) or millions (M). The graph indicates a steady increase in cumulative reads completed as time progresses.]
“…bad CPUs are 6 times more likely than bad clocks…”
• Spanner is currently in production used by Google’s advertising backend F1.
• F1 previously used MySQL that was manually sharded many ways.
• Spanner provides synchronous replication and automatic failover for F1.
• Enabled F1 to specify data placement via directories of spanner based on their needs.

• F1 operation latencies measured over 24 hours

<table>
<thead>
<tr>
<th>operation</th>
<th>latency (ms)</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
<td>std dev</td>
</tr>
<tr>
<td>all reads</td>
<td>8.7</td>
<td>376.4</td>
</tr>
<tr>
<td>single-site commit</td>
<td>72.3</td>
<td>112.8</td>
</tr>
<tr>
<td>multi-site commit</td>
<td>103.0</td>
<td>52.2</td>
</tr>
</tbody>
</table>
12.3 Summary

- Multi-version, scalable, globally distributed and synchronously replicated database.
- Key enabling technology: **TrueTime**
  - Interval-based global time
- First system to distribute data at global scale and support externally consistent distributed transactions.
- Implementation keypoints: integration of concurrency control, replication and 2PC.