Distributed Systems, Why?

• Or, why not 1 computer to rule them all?
  • Failure => Fault Tolerance
  • Limited computation/storage => Scalability
  • Physical location => Availability, Low Latency

Distributed Systems Goal

• Service with higher-level abstractions/interface
  • e.g., database, programming model, ...
  • Hide complexity - Do “heavy lifting” so app developer doesn’t need to
    • Reliable (fault-tolerant)
    • Scalable (scale-out)
    • Strong guarantees (consistency and transactions)

• Efficiently
  • Lower latency (faster interactions, e.g., page load)
  • Higher throughput (fewer machines)
What We Learned
(Much of it at least, at a very high level)

Network communication
- How can multiple computers communicate?
- Networking stack solves this for us!
- We use it to build distributed systems, relying on the guarantees it provides.

Remote Procedure Calls
- Additional layer on top of networking stack
- At least once – dealing with failures!
- At most once – ensuring correctness despite concurrency and failures

Time, logical clocks
- Concurrency!
- Real time often inadequate for distributed systems?
- Lamport clocks: $A \rightarrow B \Rightarrow LC(A) < LC(B)$
- Vector clocks: $A \rightarrow B \Leftrightarrow VC(A) < VC(B)$
Eventual Consistency, Bayou

- Favor **availability** above all else
  - e.g., disconnected Dropbox operation
- Eventual consistency
- Bayou system design
  - Operation log (logical, not physical, replication)
  - Causal consistency from log propagation and Lamport timestamps

P2P Systems & DHTs

- Efficiency of various designs
- Goal: **scale** lookup state, lookup computation, storage; **fault tolerant**
- Scale lookup state, lookup computation w/ Chord
- Scale storage with sharding
- Fault tolerance through replication, robust protocols

Dynamo

- Favor **availability** above all + **scalable** storage
- Eventual consistency (really eventual)
- Zero-hop DHT on top of data sharded with consistent hashing
  - Virtual nodes enable better load balancing (improves **throughput**), but design to still ensure fault tolerance

So far...

- Can build systems that are fault tolerant, scalable, provide low latency, highly available
- But...
- Weak guarantees
### Strong Guarantees + Fault Tolerance

- **Linearizability**: acts just like 1 machine processing requests 1 at a time!

- **Replicated state machines**:
  - Log of operations, execute in order
  - Primary-backup (and VM-FT)
    - Special mechanism for failure detection
    - React to failure
  - Paxos, RAFT
    - Built in failure detection using quorums (f+1 out of 2f+1)
    - Mask non-leader failure

### Impossibility Results Guide Us

- **CAP**: Must choose either availability of all replicas or consistency between replicas

- **PRAM**: Must choose either low latency of operations or consistency between replicas
Availability + Low Latency + Scalability + Stronger Guarantees

- COPS provides causal consistency
  - Strongest guarantees impossible w/ low latency
  - Like a scalable Bayou

- Sharding to scale storage within a datacenter
- Geo-replicate data across datacenters
  - Replication and sharding!

- New protocols for replicating writes between replicas and reading data
  - Distributed protocols w/ work on only some machines in each replica for scalability
  - Consistently reading data across shards required transactions

<table>
<thead>
<tr>
<th></th>
<th>Fault Tolerant</th>
<th>Scalable</th>
<th>Highly Available &amp; Low Latency</th>
<th>Guarantees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bayou</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>causal</td>
</tr>
<tr>
<td>Dynamo</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>eventual</td>
</tr>
<tr>
<td>Paxos/RAFT</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>linearizability</td>
</tr>
<tr>
<td>COPS</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>Causal &amp; read-only txns</td>
</tr>
</tbody>
</table>

Strong Guarantees + Scalability

- Strict Serializability: acts just like 1 machine processing requests 1 at a time with transactions across shards

- Atomic Commit w/ 2PC

- Concurrency control
  - 1 Big Lock: No concurrency
  - 2PL: Growing phase then shrinking phase
  - OCC: Assume you will succeed, only acquire locks during 2PC

<table>
<thead>
<tr>
<th></th>
<th>Fault Tolerant</th>
<th>Scalable</th>
<th>Highly Available &amp; Low Latency</th>
<th>Guarantees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bayou</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>causal</td>
</tr>
<tr>
<td>Dynamo</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>eventual</td>
</tr>
<tr>
<td>Paxos/RAFT</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>linearizability</td>
</tr>
<tr>
<td>COPS</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>Causal &amp; read-only txns</td>
</tr>
<tr>
<td>2PL</td>
<td>no</td>
<td>yes</td>
<td>-</td>
<td>strict serializability</td>
</tr>
</tbody>
</table>
Strong Guarantees + Scalability + Fault Tolerance

- Google’s Spanner
  - Sharding to scale storage
  - Paxos for fault tolerance
  - 2PL + 2PC for read-write transactions
    - Strict serializability
    - Scalable processing ... mostly

- So many reads, make read-only txns efficient!
  1. Strictly serializable read-only transactions that block, but do not acquire any locks
  2. Stale read-only transactions that do not even block

- Enabled by TrueTime
  - TrueTime gives bounded wall-clock time interval
  - Commit wait ensures a transaction completes after its wall-clock commit time

Strong Guarantees + Scalability + Low Latency?

- SNOW is impossible for read-only transactions

- Must choose either the strongest guarantees (Strict Serializability & Write transactions) or the lowest latency (Non-blocking & One Round)

- PRAM / CAP are for replication
  SNOW is for sharding

<table>
<thead>
<tr>
<th></th>
<th>Fault Tolerant</th>
<th>Scalable</th>
<th>Highly Available &amp; Low Latency</th>
<th>Guarantees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bayou</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>causal</td>
</tr>
<tr>
<td>Dynamo</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>eventual</td>
</tr>
<tr>
<td>Paxos/RAFT</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>linearizability</td>
</tr>
<tr>
<td>COPS</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>causal &amp; read-only txns</td>
</tr>
<tr>
<td>2PL</td>
<td>no</td>
<td>yes</td>
<td>-</td>
<td>strict serializability</td>
</tr>
<tr>
<td>Spanner (stale-read)</td>
<td>yes</td>
<td>yes</td>
<td>no (yes)</td>
<td>strict serializability (stale)</td>
</tr>
</tbody>
</table>

Now You Can!

- Build systems that are fault tolerant, scalable, provide low latency, highly available
  - + stronger guarantees, but not the strongest

- OR

- Build systems that are fault tolerant, scalable, and provide the strongest guarantees
Making Systems Faster

- Exploiting many types of parallelism in Facebook's Streaming Video Engine
- Reasoning about the performance of distributed systems using a mental model

Let's See It In Action

Client → Frontend Server

Inside the Datacenter

Web Tier

Executes frontend, application code

Fault Tolerance?

Scalability?

Storage Tier

Stores state, provides …
Application Code Reads/Writes to the Storage Tier

Facebook page load has 1000s of reads, chains of sequential reads dozens long [HotOS '15]

Scalable Storage is Sharded and Geo-Replicated

So Much Concurrency!

So Many Failures!
Each Backend System is a Distributed System

- But with different tradeoffs and designs depending on use
  - LIKE count?
    - Eventually consistent storage system
  - User Password?
    - Strongly consistent storage system

Each Backend System is a Distributed System

- Search results
  - Use precomputed index, precomputed with MapReduce, or a more efficient, specialized system
- Trending hashtags
  - Use a stream processing system to continuously update computation about what is most popular

Distributed Systems on …
More Systems in the Spring?!

Thanks!

- COS 375 - Computer Architecture & Organization
  - David August

- COS 461 – Computer Networks
  - Mike Freedman