Distributed Systems Engineering

> Author: Guanzhou (Jose) Hu 胡冠洲 @ MIT 6.824
>
> Teachers: Robert Morris

Many high-level theory about distributed computer systems have been covered in the course note of CS290K. For this note, I will only pick up some highlights as a complement.

Distributed Systems Highlights

Essential of distributed systems engineering is to *hide details* of the reality of distribution. However, that may sometimes limit what you can do (e.g., the MapReduce model). This is the **core tradeoff**.

We want *scaling*. Scaling out is good, but that not always solves the problem! Scaling also brings many new problems!

Threads & Concurrency

We are particularly interested in using **threads** in distributed systems (combined *event-driven* programming) because:

1. *I/O concurrency* (disk IO, networking IO, waiting for remote response, ...)
2. *Parallelism*
3. Sometimes convenient, especially for background/periodic timing or checking

Challenges of using threads arise because they must be *synchronized* over shared memory to avoid *race*s. Correct programming of concurrent events heavily rely on the language & hardware support we are using. A good candidate for concurrent programming is the Go language, because that is what it is designed to do. Go's memory model particularly demonstrates the importance of correct synchronization in multi-threading.

Fault Tolerance

Fault tolerance is one main reason behind distributed systems. Key property is to make the system still *useable* despite some classes of failures. The hard part is to still maintain the following ideal properties:

- Strongly *consistent*
- *Transparent*
- *Efficient*

To overcome *Fail-stops*, we use multiple replicas:

- Direct state transfer: periodically mirroring the whole machine's state;
- **Replicated state machines**: just sending external events
  - Flavors:
    - *Primary/Backup Replication*, e.g., VMware FT
    - *Consensus algorithms* such as Paxos, Multi-Paxos, Raft, ...;

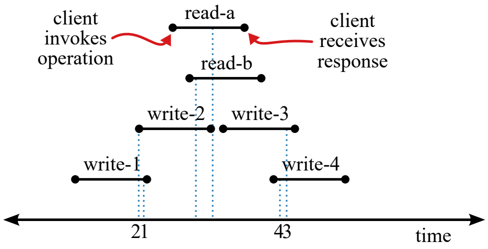
- Need to worry about:
    1. How abstractive is the state
    2. How close is the synchronization: must ensure primary $\geq$ backup $\geq$ clients
    3. Cut-over mechanism: no way to guarantee no duplicate outputs on cut-over
    4. *Split-brain* situations
    5. Building new replicas

All replication-based FT techniques can only guarantee to cope with fail-stops, not *Byzantine* failures or SW/HW bugs.

Consistency Models

Highlights the tradeoffs between consistency $\leftrightarrow$ performance, fault tolerance $\leftrightarrow$ functionality.

- "*Strong*": same behavior as a single server $\equiv$ **Linearizability**:
    - An example *history* of logged operations (seen from clients!):



    - Definition of linearizability: $\exists$ an *order* of R/W operations s.t.
        1. If A finishes before B starts in the history, then A must be ahead of B in the order, AND
        2. Each R sees the most recent (upto itself) W in the order
    - To prove / disprove linearizability:
        - Prove - find such an order
        - Disprove - find a dependency cycle
    - Some notes on enabling strong consistency:
        - *Replication* makes this very hard to guarantee
        - If an operation can *timeout* and client may resend, then we must avoid duplicated commands (by using, e.g., unique identifier for each operation)
- "*Weak*": looses some of the constraints; one successful example is GFS

Spectrum of consistency models (strong to weak):

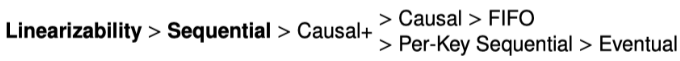


Figure from the COPS paper.

> **Eventual consistency**: reads will eventually reflect all writes, but is allowed to temporarily read stale version, and different replicas may temporarily see different data and in different order
>
> - Is a common design choice for web applications to achieve *local writes* + background pushes, by using last_writer_wins with *Lamport Clocks* (since wallclock is hard to synchronize among datacenters):
>   - T_{max} = the highest T seen in incoming writes from replicas;
>   - T = $\max(T_{max} + 1,\ \text{real time wallclock})$,
> - *Out-of-order anomacy* might be hard for programmers to reason about
> - Examples: AWS DynamoDB, Cassandra
>
> **Causal consistency** tries to solve the above anomacy (see the COPS paper) by encoding causal dependencies between operations, but may suffer from cascading dependency waits. Not popular in practice.

The **"CAP" theorem** states that a distributed replicated state machine cannot achieve all the following three properties at the same time:

- (Strongly) **C**onsistent
- (Always) **A**vailable
- (Network) **P**artition-tolerent

> Some systems use *caching* to improve performance, where we need to consider the problem of *cache coherence*.

> - Frangipani](https://pdos.csail.mit.edu/6.824/papers/thekkath-frangipani.pdf) is a successful design which achieves both caching and strong consistency
> - For modern web services, normally we need extremely high throughput but do not require strong consistency, thus allow a certain degree of staleness and only require eventual consistency. [Memcache(d) @ Facebook]() is a perfect example of such extreme high-load NoSQL KVStore design

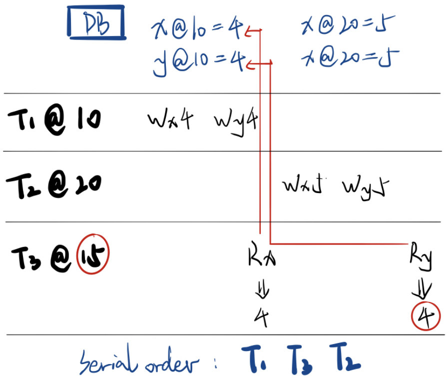
Distributed Transactions

Distributed transactions solve different problems from consensus algorithms. It is used for a data-sharded scenario (e.g., banking database) where we want to issue *different* requests to different instances (e.g., one instance debiting account A and another instance crediting account B for the same amount). A transaction consists of several *records*.

Distributed transactions $=$ Concurrency control $+$ Atomic commits.

The **"ACID" principle** states:

- **A**tomic: all or none, will not commit partial result
- **C**onsistent: called *external consistency* in database scenarios - obeys application-specific invariants
- **I**solated $\equiv$ **Serializability** - transactions cannot see each other's intermediate results but only complete (committed) transaction results
    - Formally, $\exists$ a serial order of executions of transactions that yields the actual result



    - If we wanna optimize the performance of Read-Only transactions (i.e., avoid 2PC for RO transactions):
        - Using *multi-version DB* & *timestamps*, can be achieved by **snapshot isolation**: RW transactions stamped commit time, and RO transactions stamped start time, then when a RO transaction starts a read, all the reads in it get the latest version that is no newer than its timestamp
        - *Time synchronization* is a fundamental issue that must be solved for taking such timestamps
    - Different from the notion of linearizability as linearizability is for a replicated single object and serializability is for transactions among shards
- **D**urable: persistently stored once committed

Concurrency control models:

- *Pessimistic concurrency control*: lock a record before using it
    - Conflicts cause delay; Faster when conflicts are frequent
    - Examples: Simple locking, Two-Phase Locking (2PL)
- *Optimistic ocncurrency control* (OCC): use w/o locking, leaving commits to check serializability
    - Conflicts cause abort (+ retry)
    - Faster when conflicts are rare

Atomic commits are typically done by **Two-Phase Commit** (2PC). [READ HERE](). The major disadvantage of 2PC is that the system stalls and is unavailable when a single participant crashes right after responding "YES" to a "PREPARE". 3PC allows participants to commit when the coordinator crashes, but it assumes bounded network delay. Under practical network partition scenarios, 3PC does not guarantee atomicity, thus not very interesting.

> We could make each instance a replicated state machine over a consensus algorithm for better availability. See Spanner & FaRM case studies below.

Other Case Studies

The Go Programming Language

Go-lang, developed by Google, is really powerful in handling *concurrency* and *communication*. It is designed to do that. Worth to mention that Go adopts *Garbage Collection* (GC), which greatly eases programming with only some tiny performance degrade (if GC is highly optimized).

- Go-lang: [official website]
- Go RPC package [document]
- Go memory model [document], particularly the definition of "*happens before*"

MapReduce (Lab 1); Spark

Classic distributed computing model which makes it a lot easier for users to utilize distributed computing resources, but limits the freedom of what programmers can do.

- MapReduce [paper]

Also see Spark & RDD for keeping intermediate results in memory and using the *lineage* graph for fault-tolerance:

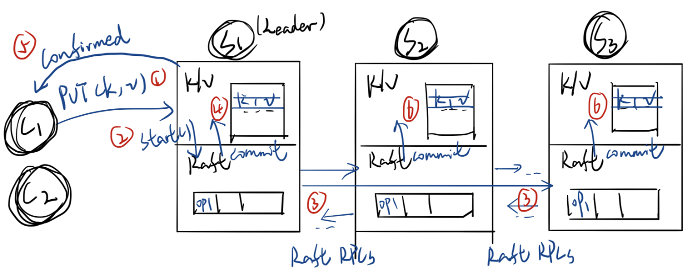
- Spark [paper]

Consensus Algorithm - Raft (Lab 2~4)

A modern & practical *consensus algorithm* which guarantees *strong consistency* over a distributed system with instance fail-stops, networking delay, and communication failures.

- Consensus algorithms solve *split-brain* problems by using the idea of **Majority Vote**: Any progress made requires a majority's (also called a *quorum*) agreement, and then any majority must *overlap* with the previous majority!
- Extend version of [paper], especially its Figure 2
- My [personal notes] on Paxos, Multi-Paxos, & Raft
- Should pay special attention to referencing outer states, because those might have been mutated under concurrent settings
- Original Raft might apply a command multiple times - *Non-idempotent* client commands must be specially handled (Section 8); correctness of Raft is defined as linearizability, thus this issue must be solved by the client
- Many performance optimizations...

Graphical workflow illustration of a service deployment over Raft, when no failures occur:



> For performance, we will want the workload to be *sharded*.
>
> - Check lab 4
> - When an operation requires atomicity across data in different shards, we must use *distributed transactions*, where the involved shards are participants. E.g., debiting Alice $5 and crediting Bob $5 at the same time
> - Also see Spanner and FaRM below for modern solutions to sharding
>
> All technical details matter when implementing a distributed algorithm - remember your failure on the midterm.

ZooKeeper over Zab

Zab is another consensus algorithm, similar to Raft. ZooKeeper is a general-purpose *coordination service* built upon Zab.

- ZooKeeper [paper]
- ZooKeeper is attractive due to:
  - It is a standalone general-purpose coordination service that is really independent of client semantics
  - Looses linearizability (can serve stale data, but guarantees per-client linearizability) and brings read performance acceleration

Chain Replication (CR)

Chain Replication is a very different approach from consensus algorithms.

- Chain Replication (CR) paper
- Chain Replication with Apportioned Queries (CRAQ) paper
- CRAQ is attractive due to:
    - It guarantees strong consistency and meanwhile enables read from any replica
    - Tradeoff is that now replication is sent through a chain of servers instead of concurrently sending to all followers $\Rightarrow$ background latency $\uparrow$

Amazon Aurora DB

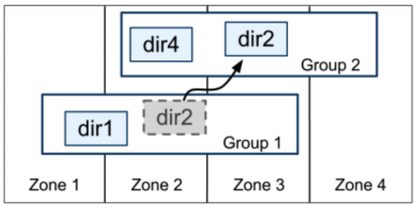
Good design of storage-separated cloud database.

- Aurora paper
- "Logs are the database": offloads replication overhead to separate storage cluster

Google Spanner

Successful design of a global-scale distributed database.

- Spanner paper

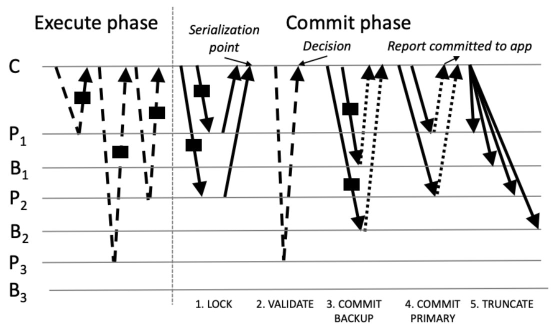


- Combines 2PC with replicated state machines:
    - Higher-level is sharding + 2PC
    - Using Paxos to replicate the coordinator & participants of distributed transactions (into a Paxos *Group* as in the figure above), to overcome the availability issues of 2PC (because each participant is now stronger against fail-stops)
- Introduces a novel *TrueTime* API which uses actual clock timestamps with a *bounded uncertainty interval*, to enable *snapshot isolation*

Microsoft FaRM

Explores optimistic concurrency control in distributed transactions.

- FaRM paper



- Similar to Spanner on the choice of 2PC over replication (but here primary+backup instead of Paxos groups)
- Not for geographically distributed database - Instead, pursues extremely high performance
    - Intra-datacenter RDMA smartNICs + kernel bypassing
    - Data fits in non-volatile RAM (NVRAM) with backup battery
- Good example of OCC:
    - Reads w/o locking & directly from target shard's memory
    - Writes are buffered locally
    - Lock + Validation when writes commit
    - Aborts a write when there are conflicts (seeing version number updates / others locked the object)

- Applications must handle aborts and retries

Worth thinking: is it possible to implement distributed transactions only using one-sided RDMA? FaRM enables RDMA reads w/o locking, but is still an open question for RDMA writes.

Blockchain + P2P Systems

"Hashed chain of history" + *“Decentralized broadcasting”*. See my 6.829 course notes.

- *Bitcoin*: Hash power contest to prevent *private double spending* (*Sybil attacks*)
- *Blockstack*: Re-build the Internet naming infrastructure over blockchain; paper