Lec 1: Introduction

Tags

Why Distributed System

- Parallelism
- Fault Tolerance
- Physical reason
  - e.g. computers in different banks
- Security / Isolation

Why Distributed System is hard

- concurrency
- partial failure: unexpected failure patterns
  - partial failure
  - network failure
- High Performance

Labs

- MapReduce
- Raft: fault tolerant techniques
- K/V Server
- Sharded K/V Service

Infrastructure

- storage
- communication
- computation

goal: abstraction, build an interface that looks to an application as if it's like a non-distributed system

Implementation

- RPC: remote procedure call,
- threads: multi-core computer, structuring concurrent operations
- concurrency control: locks

Performance

- scalability - 2x computer → 2x throughput
  - add more web servers for scaling, but when there are numerous web servers, db becomes the bottleneck

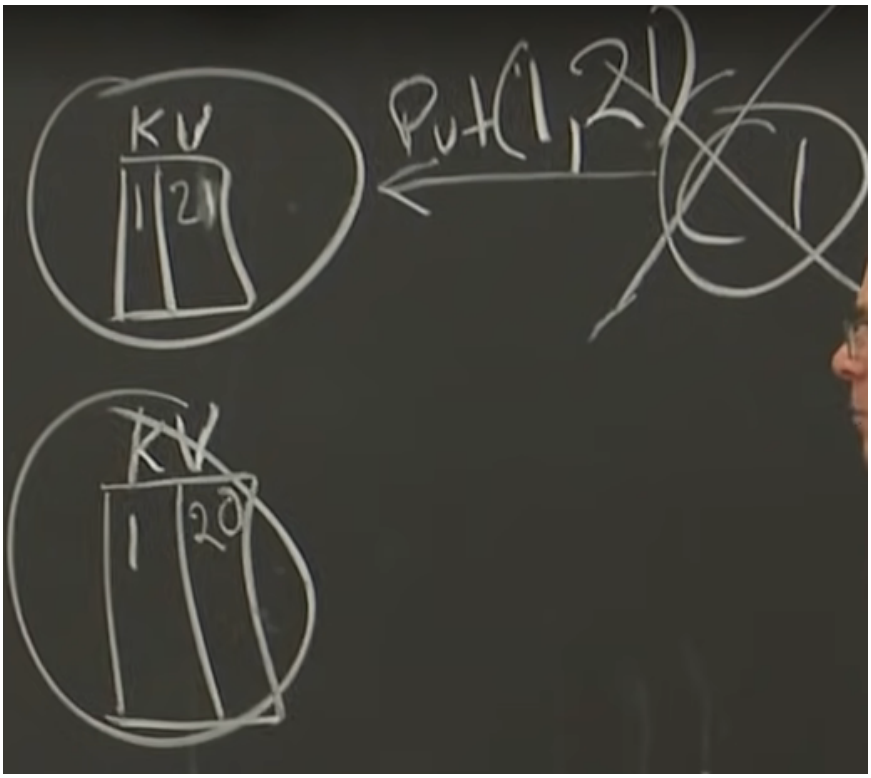
Fault Tolerance

- Inspiration
  - 1 computer, 1 year has one fault
  - 1000 computers, 3 faults per day
- Availability
  - some systems under some failures will keep operating
    - e.g. two replicated servers, one down, the other should continue operating
- Recoverability
  - something goes wrong, after the repair, working as nothing went wrong before

- Solutions:
  - non-volatile storage
    - hard-drive, ssd, to store the log or state of the system when the system is down
  - replication
    - two servers, each with identical data
      - drift out sync
    - we want the replica to have independent failure probability
      - e.g. putting both replicas in the same machine room is not a good idea as if the power cables disconnected, both replicas are dead
      - different cities, even different parts of the world, but communication becomes expensive.

Consistency

- Important KV operations
  - PUT(k, v)
  - GET(k) → v



In this case, server 2 is down, so if the user tries to read the data with key 1, the values are not consistent

- Strongly Consistent vs Weakly Consistent
  - Strong: get most recent values
    - very expensive, a lot of communications to consult every copies to find the most recent values
  - Weak: doesn’t guarantee getting the most recent value
    - allow stale read of old values

MapReduce

- Google, 2004
  - huge computation with terabytes of data
    - e.g. creating index of the content of the web, analyzing the link structure of the entire web to identify the most important web pages

    - used to give these tasks to clever engineers, using a lot of computers for these computation, kind of one-off
    - MapReduce becomes a framework that hides the details of distributed system, available for non specialist to understand
- Map(k, v)
    - call map function on each of the inputs, parallelism available, produce a list of key-value pairs (intermediate outputs)
    - e.g. word count: map each word as a key with value 1, a → 1, a → 1, b → 1, c → 1, b → 1
    - Implementation:
        - split v into words
            - for each word w:
                - emit(w, “1”)
        - k is the filename, we typically just ignore it
- Reduce(k, v)
    - collect instances from all maps of each keyword, hand them to reduce functions
    - e.g. word count: a → 2, b → 2, c → 1
    - emit(len(v))
- Low-levels
    - master server
        - know how many inputs in total
    - worker servers
        - know all the map reduce
        - call map
        - write data to files on the local disk
        - at the end of map phase
- Input and Outputs

    - GFS: Google File System
        - Splits up large files into 64 MB chunks, distribute evenly to Google file server
        - mapreduce, 1000 workers read from 1000 GFS, great total read thoroughput

What happened from input to the map function?

- work process talk across the correct network to the correct GFS servers to store its part of input
- avoid using the network, since root ethernet switch could be the bottleneck (50 MB/s for each machine)
    - solution: run GFS and MapReduce Workers on the same machine