Map Reduce & GFS

CS 475, Spring 2018
Concurrent & Distributed Systems
Review CAP Theorem

• Pick two of three:
  • Consistency: All nodes see the same data at the same time (strong consistency)
  • Availability: Individual node failures do not prevent survivors from continuing to operate
  • Partition tolerance: The system continues to operate despite message loss (from network and/or node failure)

• You can not have all three, ever*
  • If you relax your consistency guarantee, you might be able to guarantee THAT…
Review: CAP Theorem

- C+A: Provide strong consistency and availability, assuming there are no network partitions
- C+P: Provide strong consistency in the presence of network partitions; minority partition is unavailable
- A+P: Provide availability even in presence of partitions; no strong consistency guarantee
Relaxing Consistency

• We can relax two design principles:
  • How stale reads can be
  • The ordering of writes across the replicas
Eventual Consistency

- Allow stale reads, but ensure that reads will **eventually** reflect the previously written values
  - Eventually: milliseconds, seconds, minutes, hours, years…
- Writes are NOT ordered as executed
  - Allows for conflicts. Consider: Dropbox
- Git is eventually consistent
Announcements

• HW4 Due Friday
• Project is out!!!
  • http://www.jonbell.net/gmu-cs-475-spring-2018/final-project/
  • (Hey, it could be worse)
• Today:
  • Big data problems
• Additional readings:
  • GFS, MapReduce papers
More data, more problems

• I have a 1TB file
• I need to sort it
• …My computer can only read 60MB/sec
• …
• …
• …
• 1 day later, it’s done
More data, more problems

• Think about scale:
  • Google indexes ~20 petabytes of web pages per day (as of 2008!)
  • Facebook has 2.5 petabytes of user data, increases by 15 terabytes/day (as of 2009!)
Distributing Computation

QUICK, ROBIN

TO THE CLOUD
Distributing Computation

- Can't I just add 100 nodes and sort my file 100 times faster?
- Not so easy:
  - Sending data to/from nodes
  - Coordinating among nodes
  - Recovering when one node fails
  - Optimizing for locality
  - Debugging
Distributing Computation

• We begin to answer
  • 1. How do we store the data?
  • 2. How do we compute on this data?
GFS (Google File System)

- Google apps observed to have specific R/W patterns (usually read recent data, lots of data, etc)
- Normal FS API (POSIX) is constraining (consider: CFS contains a ton of annoying glue to make it work)
- Hence, Google made their own FS
GFS

- Hundreds of thousands of regular servers
- Millions of regular disks
- Failures are normal
  - App bugs, OS bugs
  - Human Error
  - Disk failure, memory failure, network failure, etc
- Huge number of concurrent reads, writes
GFS Workload

• (Relatively) small total number of large files (>100MB) - millions
• Large, streaming reads (reading > 1MB at a time)
• Large, sequential writes that always append to end of a file
• Multiple clients might append concurrently
GFS Design Goals

- Unified FS for all google platforms (e.g. gmail, youtube)
- Data + system availability
- Graceful + transparent failure handling
- Low synchronization overhead
- Exploit parallelism
- High throughput and low latency
GFS Architecture
GFS Architecture

- Single master server (RSM replication to backups)
  - Holds all metadata (in RAM!) - namespace, ACL, file-chunk mapping
  - In charge of migrating chunks, GC’ing chunks
- Data stored in 64MB chunks each with some ID
  - Compare to EXT-4’s 4KB block
- Thousands of chunk servers
  - Chunks are replicated
  - Chunk servers don’t cache anything in RAM, store chunks as regular files
GFS Client

- Makes metadata requests to master server
- Makes chunk requests to chunk servers
- Caches metadata
- Does not cache data (chunks)
  - Google’s workload (streaming reads, appending writes) doesn’t benefit from caching, so why bother with consistency nightmare
GFS Reads

• Client asks master for chunk ID, chunk version number, and location of replicas given a file name
• By default, GFS replicates each chunk to 3 servers
• Client sends read request to closest (in network topology) chunk server
GFS Writes

- Client asks master for replicas storing a chunk (one is arbitrarily declared primary)
- Client sends write request to all replicas
- Each replica acknowledges write to primary replica
- Primary coordinates commit between all of the replicas
- On success, primary replies to client
GFS Chunk Primaries

- There needs to be exactly one primary for each chunk
- GFS ensures this using leases
  - Master selects a chunk server and grants it a lease
  - The chunk server holds the lease for $T$ seconds, and is primary
  - Chunk server can refresh lease endlessly
  - If chunk server fails to refresh it, falls out of being primary
- Like a lock, but needs to be renewed (like with a heart beat)
GFS Consistency

• Metadata changes are atomic. Occur only on a single machine, so no distributed issues.
• Changes to data are ordered as arbitrarily chosen by the primary chunk server for a chunk
GFS Summary

• Limitations:
  • Master is a huge bottleneck
  • Recovery of master is slow
• Lots of success at Google
• Performance isn't great for all apps
• Consistency needs to be managed by apps
• Replaced in 2010 by Google's Colossus system - eliminates master
Distributing Computation

• Lots of these challenges re-appear, regardless of our specific problem
  • How to split up the task
  • How to put the results back together
  • How to store the data (GFS)
• Enter, MapReduce
MapReduce

- A programming model for large-scale computations
  - Takes large inputs, produces output
  - No side-effects or persistent state other than that input and output

- Runtime library
  - Automatic parallelization
  - Load balancing
  - Locality optimization
  - Fault tolerance
MapReduce

- Partition data into splits (map)
- Aggregate, summarize, filter or transform that data (reduce)
- Programmer provides these two methods
MapReduce: Divide & Conquer

Big Data (lots of work)

Partition

w1
worker
r1

w2
worker
r2

w3
worker
r3

w4
worker
r4

w5
worker
r5

Combine

Result
MapReduce: Example

- Calculate word frequencies in documents
- Input: files, one document per record
  - **Map** parses documents into words
    - Key - Word
    - Value - Frequency of word
  - **Reduce**: compute sum for each key
MapReduce: Example

Each line goes to a mapper

Input 1:
apple orange mango
orange grapes plum

apple orange mango
orange grapes plum

Input 2:
apple plum mango
apple apple plum

apple plum mango
apple apple plum

Map splits key->value

apple, 1
orange, 1
mango, 1

orange, 1
grapes, 1
plum, 1

apple, 1
plum, 1
mango, 1

apple, 1
apple, 1
plum, 1

To reduce
MapReduce: Example

From Map

- apple, 1
- orange, 1
- mango, 1
- orange, 1
- grapes, 1
- plum, 1
- apple, 1
- plum, 1
- mango, 1

Sort, shuffle

- apple, 1
- apple, 1
- apple, 2
- grape, 1
- mango, 1
- mango, 1
- orange, 1
- orange, 1
- plum, 1
- plum, 1

Reduce

- apple, 4
- grape, 1
- mango, 2
- orange, 2
- plum, 3

Final Output

- apple, 4
- grape, 1
- mango, 2
- orange, 2
- plum, 3
MapReduce Applications

- Distributed grep
- Distributed clustering
- Web link graph traversal
- Detecting duplicate web pages
MapReduce: Implementation

- Each worker node is also a GFS chunk server!
MapReduce: Scheduling

- One master, many workers
- Input data split into $M$ map tasks (typically 64MB ea)
- $R$ reduce tasks
- Tasks assigned to works dynamically; stateless and idempotent -> easy fault tolerance for workers
- Typical numbers:
  - 200,000 map tasks, 4,000 reduce tasks across 2,000 workers
MapReduce: Scheduling

- Master assigns map task to a free worker
  - Prefer "close-by" workers for each task (based on data locality)
  - Worker reads task input, produces intermediate output, stores locally (K/V pairs)
- Master assigns reduce task to a free worker
  - Reads intermediate K/V pairs from map workers
  - Reduce worker sorts and applies some reduce operation to get the output
Fault tolerance via re-execution

• Ideally, fine granularity tasks (more tasks than machines)
• On worker-failure:
  • Re-execute completed and in-progress map tasks
  • Re-executes in-progress reduce tasks
  • Commit completion to master
• On master-failure:
  • Recover state (master checkpoints in a primary-backup mechanism)
MapReduce in Practice

• Originally presented by Google in 2003
• Widely used today (Hadoop is an open source implementation)
• Many systems designed to have easier programming models that compile into MapReduce code (Pig, Hive)
Hadoop: HDFS
HDFS (GFS Review)

• Files are split into blocks (128MB)
• Each block is replicated (default 3 block servers)
• If a host crashes, all blocks are re-replicated somewhere else
• If a host is added, blocks are rebalanced
• Can get awesome locality by pushing the map tasks to the nodes with the blocks (just like MapReduce)
Hadoop + ZooKeeper

- Hadoop uses ZooKeeper for automatic failover for HDFS
- Run a ZooKeeper client on each NameNode (master)
- Primary NameNode and standbys all maintain session in ZK, primary holds an ephemeral lock
- If primary doesn’t maintain contact it session expires, triggering a failure (handled by the client)
Hadoop + ZooKeeper

Hadoop + ZooKeeper

ZK Server

ZK Client
NameNode
Primary

ZK Server

ZK Client
NameNode
Secondary

DataNode

DataNode

DataNode

DataNode

DataNode

DataNode

DataNode

DataNode

DataNode

DataNode

DataNode

DataNode

DataNode

DataNode

DataNode

DataNode
Hadoop + ZooKeeper

- **ZK Server**
- **ZK Client**
- **NameNode**
- **DataNode**

**Diagram Description**:
- The diagram illustrates the interaction between Hadoop and ZooKeeper components.
- **ZK Server**: Three ZK Server nodes are shown.
- **ZK Client**: A ZK Client node is connected to each NameNode.
- **NameNode**: There are two NameNode nodes, one marked as Primary and the other as Secondary.
- **DataNode**: Multiple DataNode nodes are depicted around the NameNode nodes.

**Key Points**:
- **Primary NameNode** receives notifications that the leader is disconnected and becomes the new Primary.
- **Secondary NameNode** serves as a backup to the Primary.
- The diagram highlights the role of ZooKeeper in maintaining a quorum of NameNodes.

**Legend**:
- **timeout** indicates a wait period.
- **disconnected** signifies a connection failure.

**Inference**:
- ZooKeeper provides a reliable and scalable solution for managing Hadoop cluster configurations.
Hadoop + ZooKeeper

Note - this is why ZK is helpful here: we can have the ZK servers partitioned *too* and still tolerate it the same way.
Hadoop + ZooKeeper

• Why run ZK client in a different process?
• Why run ZK client on the same machine?
• Can this config still lead to unavailability?
• Can this config lead to inconsistency?
Hadoop Ecosystem