CSCI 1380  Day 21
How do you determine the appropriate value for heartbeat timeouts?

\[ \text{Timeout} = N \times (\text{latency} + \text{latency of marshalling} + \text{latency of app processing heartbeats}) \]

\[ \text{Timeout} = \text{func} (\text{latencies, failure scenarios}) \]
Today

Conclude Cassandra / Dynamo
GFS / MapReduce / Colossus
Kafka / Samza
Coordinator = node that checks, says which node is in charge of the key

FE

- pick random node

get metrics

changes the "IDs" of servers to balance the system

Coordinator

- Use check to redirect

Pick random node which happens to be correct

Key is stored at N nodes

- the node in charge according to check

+ plus N-1 other nodes

- failure detector

Φ

| Time taken = time receive - time send |
| the lost N hours of heartbeats |

plot the distribution of time taken

soms

[Graph]

(replaces static heartbeat timeout)

99th percentile

if N/2 becomes more or less congested:

the distribution will change & this will naturally change the timeout value

Pick the 99th percentile time taken as the heartbeat timeout

simple math

is an array of value
Good = keep making progress
Bad = no consistent view of data

There's no consistency = when the partition heals then you have conflicting data for the same key

Dynamo = application developer will write code to use vector clocks to fix conflicts (merge everything into one cart)
Cassandra = last writer wins
Cassandra/Dynamo

Partition
Membership
Consistent Hashing
Fault Tolerance
Replica

Consistent Hashing
Due to Hash Function
Uneven Distribution
"Coordinator" which
Changes Node's ID

Conflict Handling
Least-Writer Wins

Static/Local Approach
(Similar to an Approach in the Project)

Virtual Nodes
Application Specific Merge Function

Cassandra

Dynamo
Problems

1. File sizes (very small)
2. Complicate querying
3. Expensive writes
4. optimizations (not helpful)

New optimizations for metadata
New optimization to improve failure scenarios
What is Google search?

Search = large platform that archives webpages & allows for queries on these archived

Indexed format

Some processing on the archive to rank/eliminate duplicates/other

Work flow:
[ Every night Google downloads the Internet (webpages) then creates indexes & ranks them ]

Google File System (GFS):

1. Always append (never random write/change a file)
2. Huge files
3. Mostly read queries (mostly sequential reads)
files are huge = chunks/blocks should be huge

file is broken down into chunks/blocks

GFS chunk = 64MBs
paddlestore block = 64 BS

chunk size is a function of file sizes

Sequential read specific
1. prefetch & cache both data & metadata
appends in GFS (avoid locks)

1. chunk server received append request
2. chunk server decides order to process them (hopefully FIFO)
3. it returns location where append is placed to the server

You can have append without locks ⇒ the clients don’t get to choose where to append to
GFS

1. Large chunk sizes $\Rightarrow$ large files
2. Caching/prefetching $\Rightarrow$ read-heavy workloads
3. Elegant append $\Rightarrow$ append-only writes (lock-free)
4. Split metadata from data

Run on just one server without significant scale issues
Run on many servers
Summary

1. Cassandra vs Dynamo
2. GFS
   * requirements
   * Implications on design
3. Evolution of Google file systems

As a company evolves, the requirements, users, apps, & developers will change ⇒ file system will have to evolve.

Megastore

BigTable

GFSv1 → GFSv2

Applications / Users