International Institute of Information Technology, Bangalore

Distributed Computing Concepts Reading Elective

Satvik Ramaprasad | IMT2016008

December 3, 2019

Contents

1 Objective

The objective of the reading elective is to learn core distributed computing concepts. Beginning with CAP theorem which essentially states that a system cannot be fault tolerant, available and consistent at the same time, the goal is to learn how different concepts and techniques help in achieving eventually consistency.

The main concepts are:

- CAP Theorem
- Gossip Protocol
- Failure Detection and Recovery
- Consistent Hashing
- Database Sharding
- Time and ordering - Vector Clocks
- Caching
- Consensus Problem - Paxos

A secondary objective is to study how these concepts are put in practice by doing case studies. It is important to note the design considerations and trade offs that are done at the same time.

The case studies are:

- MySQL
- Cassandra
- BigTable
- DynamoDB
- Google File System

A large part of the reading elective is done by going through the course content of Cloud Computing Concepts, Part 1 offered by University of Illinois on coursera. The case studies are done by going through the official papers, documenation and other articles.

2 CAP Theorem

In the past, all systems were vertically scaled, that is when the system performance needed to be increased, the system specs was increased. More CPU and RAM was added. Additional storage were mounted to increase storage capacity. However, this became quickly infeasible as it quickly starts getting expensive and there is an upper bound in compute, memory and storage limited by technology.

Therefore systems were starting to be scaled horizontally - that is multiple computers running in parallel. This could be 10 nodes or massively distributed such as a million nodes. However, this brought its own problems as we now have to synchronize, share and coordinate across a network. When there are so many computers, failures becomes the norm rather than the exception. So can we build a perfect distributed system? The famous CAP theorem says no. The Venn diagram in Figure 2.1 shows that only two of consistency, availability and partition tolerance can be achieved at the same time.

Note: all diagrams referenced from [2].

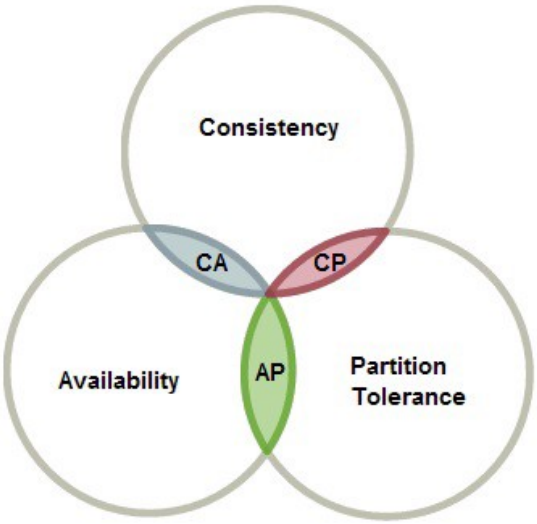

Figure 2.1: CAP Theorem Venn Diagram

2.1 C: Consistency

This condition basically means all nodes on the network see the same data state at all times. Essentially, the read operation at all times will result in the latest write. For this to be possible, all nodes should be able to communicate so that they can be in perfect synchronisation.

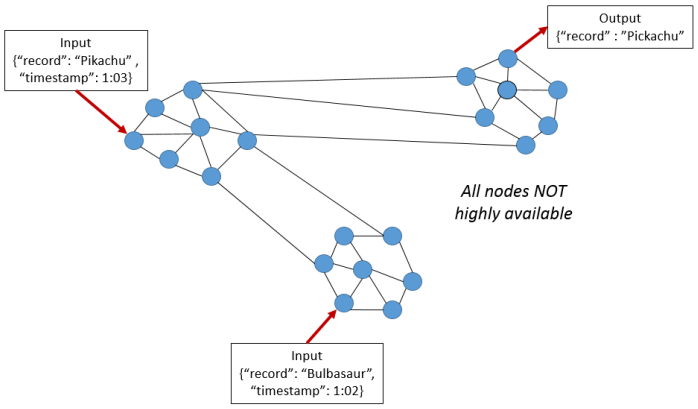

Figure 2.2: Strong Consistency sacrifices availability during network failures.

2.2 A: Availability

Available essentially means that the system will always respond even in the face of network failures or individual node failures.

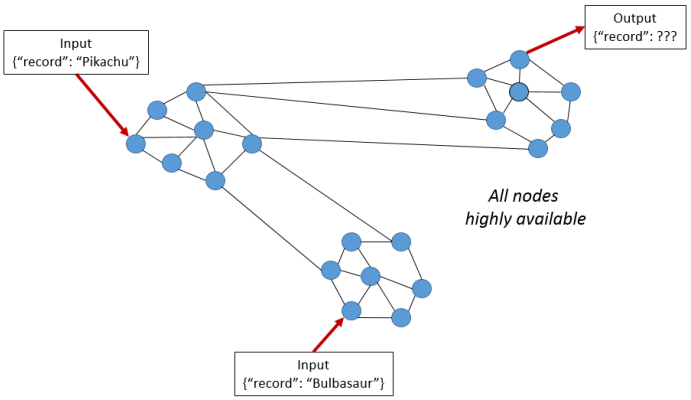

Figure 2.3: Availability guarantees that every request gets a response/failure even if it doesn’t have the correct answer.

2.3 P: Partition Tolerance

Partition tolerance basically means the system continues to run if there are network delays or failures, i.e the system is tolerant to network failures.

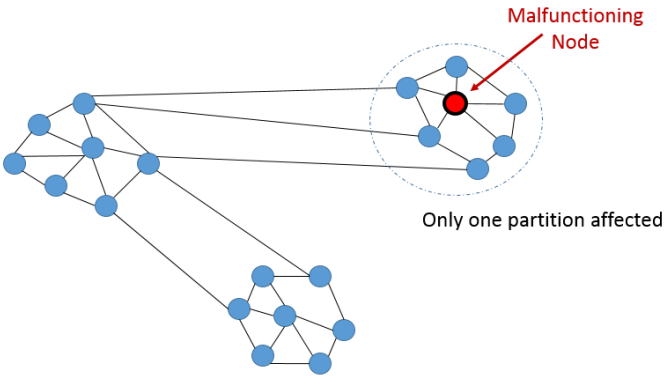

Figure 2.4: System continues to work even if sections of the network are disconnected

2.4 CAP - Analysis

The theorem states that only 2 of these 3 characteristics can be satisfied at the same time. This is a big problem as all 3 characteristics are required.

Partition tolerance is essential as data centers are separated by seas and network failures and delays are very common. A trade off has to be done between consistency and availability. A large class of systems choose availability over consistency and instead ensure eventual consistency which essentially means the system will eventually be consistent. Broadly these systems are called BASE (Basically available soft state eventual consistency) systems. Note that eventual consistency doesn't give an upper bound on how long it will take to reach consistency. However in practice it usually reaches consistency in a few seconds. Another class of systems choose consistency over availability. These systems generally run critical tasks such as financial transactions where consistency is priority.

In practice, most systems use a combination of both classes of systems. For example an E-Commerce site can use a highly available system to deliver the product details but use a highly consistent system to process orders/transactions.

The proof is rather simple can be proved by fixing any 2 of the characteristics at the same time and trying to introduce the 3rd characteristic. The full proof can be seen here [10].

3 Gossip Protocol

There are N nodes on the network and one node gets some information and it needs to disseminated through out the network efficiently minimizing network load and latency. This is also known as reliable multi-cast. There are many ways of performing this task, gossip protocol is a very popular and simple method of dissemination.

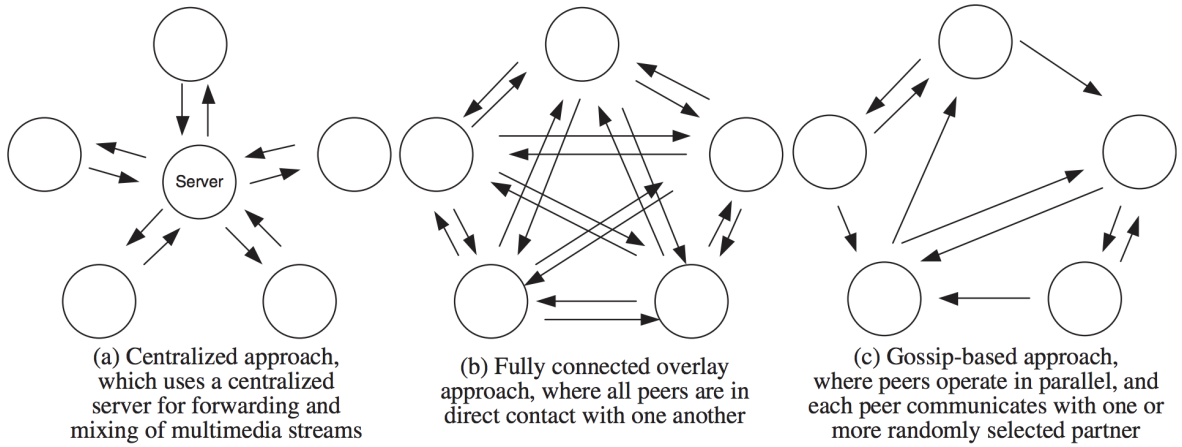

Figure 3.1: Gossip Protocol

A visual simulation of the gossip protocol can be found here - `https://flopezluis.github.io/gossip-simulator/`

References: [19] [9] [3] [8]

3.1 Simple Solution

Before we begin, we should discuss a simple solution. We can maintain a tree like structure between the nodes. When new information arrives at a node, we can start propagating the information to children and then children will propagate the information. This is very efficient, takes only log n steps and load on the network is low.

Though this is efficient in theory, it is not practical because of the following reasons

1. Nodes join and leave the network - So a tree structure cannot be maintained
2. Nodes can fail - Information dissemination will stop if a node fails, therefore the algorithm cannot rely on any node to send information to a set of nodes.
3. Network is reliable, messages can be delayed indefinitely.

3.2 Gossip Solution

The gossip solution is inspired from the way rumours and epidemics spread - which is randomly but rapidly. When a node gets a piece of information, it randomly selects some number of nodes b and sends the same information to them. Note that the nodes need not know about all the nodes. b is generally small such as a value of 2. Each node can execute the gossip protocol periodically.

There are primarily 3 variants of the protocol

- Push - Nodes push recent information
- Pull - Nodes ask for information since some time t
- Hybrid Push-Pull - Does both push and pull approaches

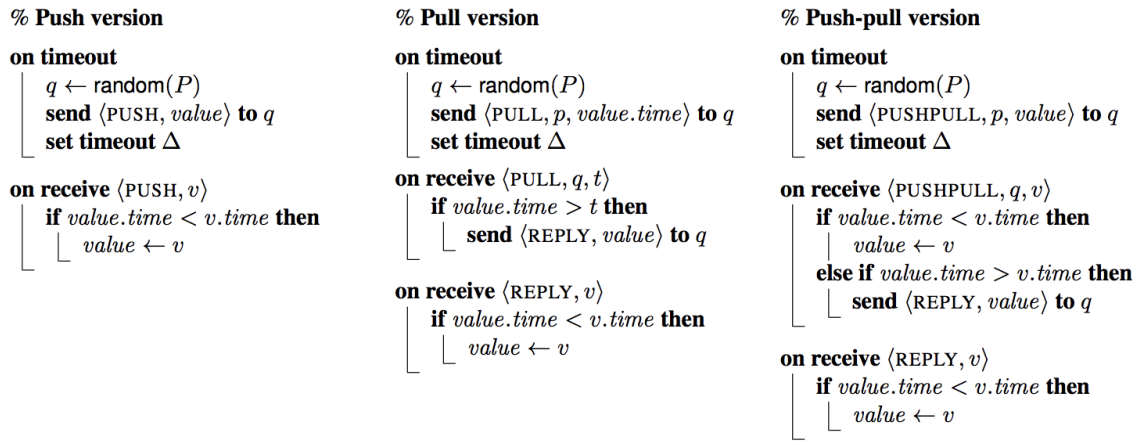

Figure 3.2: Gossip Protocol - Algorithm [8]

Mathematical analysis shows that in the beginning a push based protocol is faster and towards the end a pull based protocol is faster. Therefore, the hybrid approach even though there is more communication overhead generally performs better.

Analysis of Gossip Protocols show that it is

- **Reliable** - The protocol works even if some messages are dropped or some nodes fail. There can also be a bound on the time taken to disseminate. Therefore the protocol is reliable.
- **Low Latency** - Analyzing mathematically, we can show that it will take $O(\log(n))$ rounds to completely disseminate information. So latency is low.
- **Low network load** - Each node sends information to a few nodes in each round.
- **Fault tolerant** - The protocol works even if some messages are dropped or some nodes fail. Therefore the protocol is fault tolerant.

3.3 Topology Aware Gossip Protocol

In practice, each node disseminating randomly may not be very effective in utilizing the network bandwidth and reducing latency. This is because packets take less time to move across racks in a data center as compared to nodes between data centers. Therefore the topology of the network has to be taken into consideration when applying the gossip protocol.

More information can be found in this paper [13].

4 Failure Detection and Recovery

In modern distributed systems, failures are the norm rather than the exception. The MTBF (mean time between failures) for a single server is typically 24 months. However, the MTBF for a cluster of 100K nodes is about an hour. So the system has to be resilient to node failures.

Detecting failures quickly and reliably is very important to have systems replace the failed nodes. If failures are not detected and recovered from, it will lead to data loss, inconsistency and poor availability.

4.1 Types of failure models

Broadly, there are 3 types of failure models.

1. **Crash Stop/ Fail stop process failures** - Once Processes fail, they stop execution until time infinite and do not recover. They do not take actions that deviate from the original algorithm
2. **Crash Recovery Model** - Where processes once failed can recover and rejoin the system with a different identifier.
3. **Byzentine failure model** - Where processes can deviate from the algorithm. This typically happens when there are malicious parties in a mutually distrusting set of parties.

Crash recovery and crash stop model are similar. The Byzentine failure model while very important is not very relevant in most distributed systems as generally all nodes are managed by a single party.

4.2 Membership Lists

To solve the problem, nodes maintain a membership list. A membership list is a list which contains most of the processes that are there in the system that have

not failed yet. This list is used by several applications and algorithms such as the gossip protocol.

The list needs to be up to date if processes join, leave or fail. This has to be done over unreliable communication medium.

4.3 Failure Detector

Failure detector is a process that detects the failure of a faulty process. It then uses dissemination to update membership list. Some chracterstics of a good failure detector are.

1. **Completeness** - A failed process is detected by at least one non-faulty process.
2. **Accuracy** - When a process is detected as failed, it actually has failed (No false positives).
3. **Latency** - delay in detection of failure to be as small as possible
4. **Scale** - algorithm should scale with the number of processes. It should have low network load, no single point failures, no bottle networks and should support aribitrary simultaneous failures.

Unfortunately, achieving both completeness and accuracy perfectly is not possible.

Completeness is required because cost of missing failures is very high. Mistakenly detected failures, the failed process can re-join the system with a different process identifier.

4.4 Heartbeat Algorithms

In order to detect failures, heartbeats are regularly sent or requested between nodes. Repeatedly missed heartbeats can indicate a failed node.

Centralized Heartbeating

A node sends a periodic heart beat to some central node, the node remembers the heartbeat. If the heartbeat stops then it is marked as failed. In this method, there is a single point of failure. The load on the central node is high. Moreover, if the central node fails, no one can detect it.

Ring Heartbeating

Each node sends to at least one neighbour (1 left, 1 right). This can handle single individual failures well. However, multiple failures may go undetected. Moreover, repairing the ring or adding and removing nodes from the ring is complicated.

All to All Heartbeating

Each node Sends heartbeat to all nodes in the system - results in high network load equally on all members. However, there is no single point of failure

Gossip Style Heartbeating

This is a variant of all to all heartbeating. It is an asynchronous system in which each process maintains a list which contains process address, heart beat counter and local time (Asynchronous system).

Periodically, each node sends the entire table to some nodes randomly periodically (T-Gossip). Each node merges the list into its own list. Merging involves updating time and counter.

If no heartbeat is heard after some timeout (T-fail), the process is marked as failed. However, the entry cannot be deleted immediately as other processes may not have detected it as failed. After T-fail, the heartbeat is not gossiped. After some time (T-cleanup), the entry is deleted later.

5 Database Sharding

When a database table grows and can no longer fit in one system or we need to speed up query response times, sharding can be used. Sharding is a database architecture pattern related to horizontal partitioning — the practice of separating one table’s rows into multiple different tables, known as partitions.

Database sharding can be done with SQL as well as NoSQL databases. However, in general NoSQL databases are designed to be horizontally scalable while SQL databases are designed to be vertically scalable to provide ACID guarantees.

Note: all images referenced from [6]

Sharding involves breaking up one’s data into two or more smaller chunks, called logical shards. The logical shards are then distributed across separate database nodes, referred to as physical shards, which can hold multiple logical shards. Despite this, the data held within all the shards collectively represent an

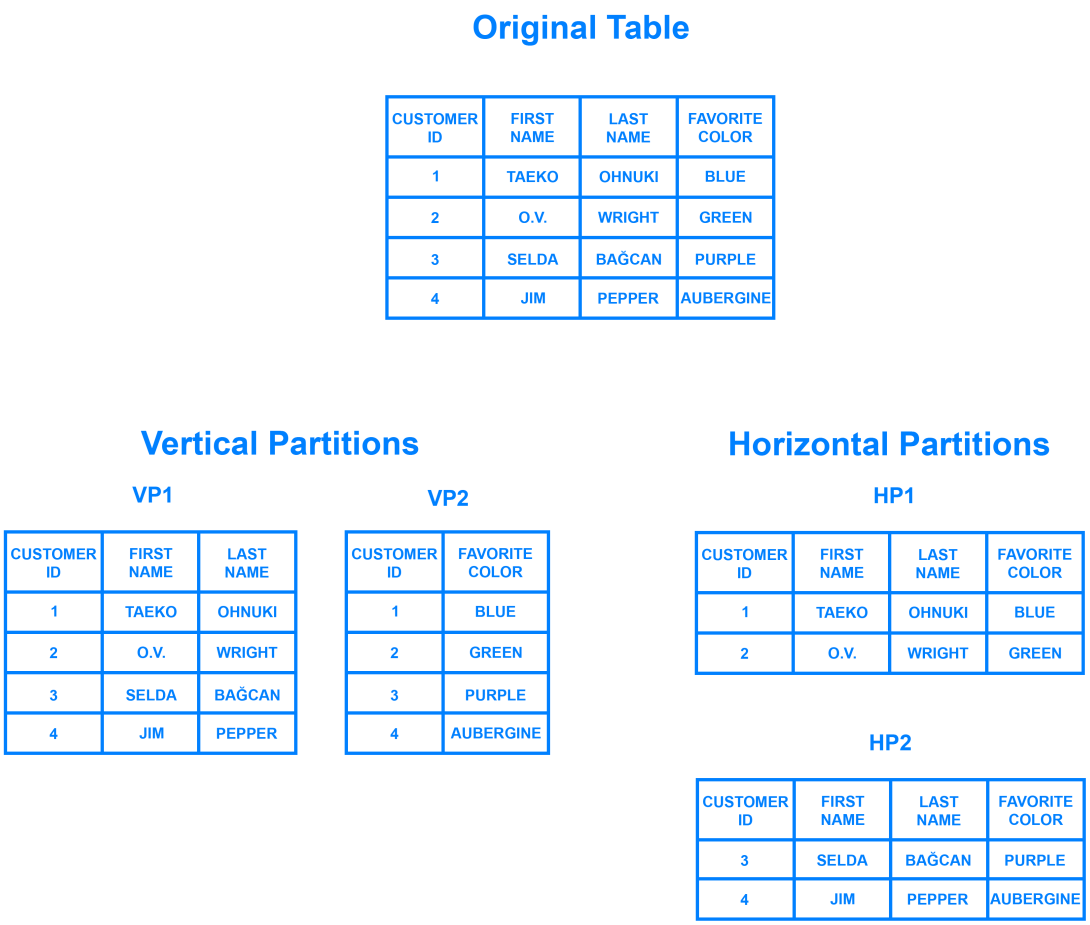

Figure 5.1: Sharding Example

entire logical dataset [6]. Database shards use a shared nothing principle, i.e. the shards generally do not share or synchronize data between other shards.

This means certain things which were very simple in traditional relational database systems becomes very hard now due to the shared nothing architecture and may have to be handled at the application level.

1. SQL Queries across shards
2. Consistency across shards
3. Joins across shards

5.1 Advantages and Disadvantages

Most of the following points are relevant to traditional relational databases. Many of the disadvantages either don't apply or are mitigated by NoSQL systems.

Advantages

1. Sharding a database can help to facilitate horizontal scaling
2. Sharded database architecture helps to speed up query response times
3. Sharding makes a system more reliable by mitigating impact of outages

Disadvantages

1. Sharded databases are very complex to setup.
2. Shards get unbalanced easily. Rebalancing is costly.
3. Once a database is sharded, it is difficult to return to an unsharded architecture.
4. Sharding isn't natively supported by many database engines.
5. Range queries and joins have to now be done in the application layer and imposes high latency and performance degradation. Foreign keys, cannot be maintained across shards as the shards share nothing between them. Therefore, many of the acid properties and advantages of a relational database breaks down.

5.2 Sharding Methods

5.2.1 Key Based Sharding

Sharding is based on the hash value of the key. The hash value of the key is calculated and then the hash is used to determine which shard to go to. One issue is if the hash function is bad and sends data to the same shard. Other possible issues are complexities in adding and removing new shards.

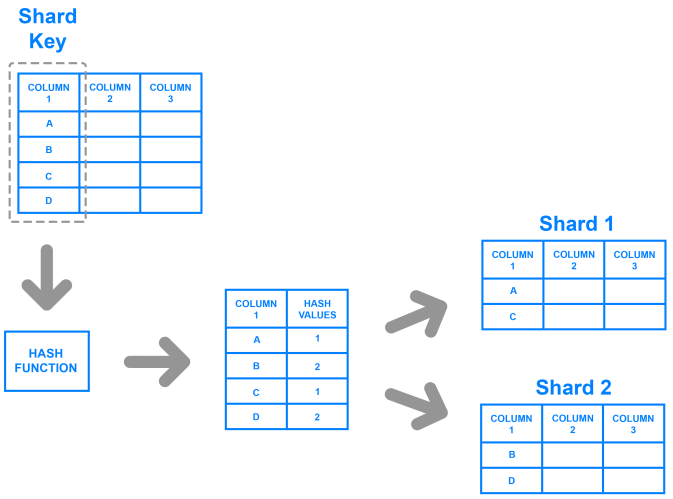

Figure 5.2: Sharding Example - Key Based

5.2.2 Range Based Sharding

Range based sharding involves sharding data based on ranges of a given value. Main advantage of this method is that it is relatively easy to implement. However, it doesn't provide any protection against uneven sharding. Also shards are more susceptible to become hot spots in range based sharding.

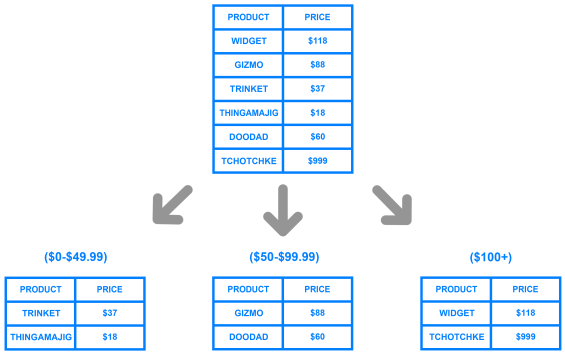

Figure 5.3: Sharding Example - Range Based

5.2.3 Directory Based Sharding

Directory based sharding is based on using a lookup table that uses a shard key to keep track of which shard holds which data. The main advantage of directory based sharding is its flexibility

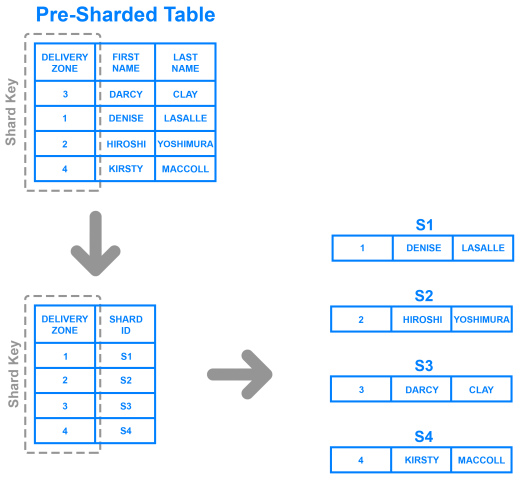

Figure 5.4: Sharding Example - Directory Based

6 Consistent Hashing

Some of the problems with sharding were

- If nodes enter and leave the network, the database has to be resharded which is very expensive.
- If a node/shard fails, which node/shard has to take over.
- If the hashing function is not good and is causing one of the shards to get heavy load, handling this scenario is difficult.

Consistent hashing helps in solving all these issues. Consistent hashing is used in several load balancing applications including database sharding.

6.1 Ring

The nodes are arranged in a ring. The ring is numbered say 1 - 1024. The nodes can be initially be arranged in a uniform manner and pegged to a number. When a query comes, the hash of the key is calculated and the query is sent to the nearest node to the right.

This helps in managing different shards. When a new node arrives it just needs to be added into the ring. When a node leaves, the next node in the ring should take over. Therefore all the data in a node needs to be replicated on the next n nodes on the ring.

This method helps mitigating the costs of a bad hashing function and allows adding and removing of nodes relatively easily. However, it is still costly and when

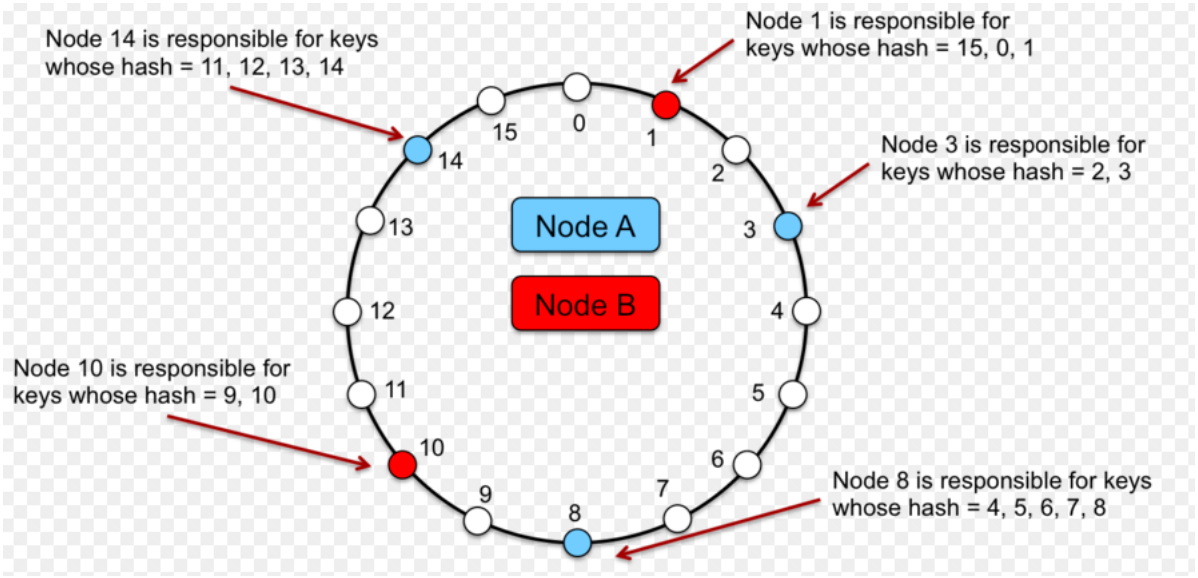

Figure 6.1: Consistent Hashing with Virtual Nodes

nodes enter and leave, the load is not very balanced.

6.2 Virtual Nodes

Instead of a node having just one number in the ring, a node can have many virtual positions on the ring. This means when a node enters or leaves, each node gets effected by only k/n records. Similarly when a node fails, its load can be balanced by several different nodes.

6.3 Complexity Analysis

Asymptotic time complexities for N nodes and K keys [5].

Action	**Classic hash table**	**Consistent hashing**
add a node	O(K)	O(K/N+log(N))
remove a node	O(K)	O(K/N+log(N))
add a key	O(1)	O(log(N))
remove a key	O(1)	O(log(N))

The $O(log(N))$ complexity for consistent hashing comes from the fact that a binary search among nodes angles is required to find the next node on the ring.

7 Caching

Caching is a high speed layer which stores a subset of data and helps increase performance, reduce load and decrease latency. Cache works be storing recently computed data or recently retrieved data. It helps by reducing network calls and in turn reduces the latency. Caching also helps in reducing database load. Cache takes advantage of the temporal and localized nature of queries.

Cache hardware is more expensive (SSD) as compared to DB servers. A small amount of data should be stored in cache as otherwise cache also gets slow and expensive.

When the query is found in the cache, its called cache hit, when its not found its called cache miss. When there are lots of cache hits, the cache is considered as hot. If there are a lot of cache misses, the cache is considered to be trashing and is degrading for system performance.

Reference: [7]

7.1 Cache Eviction Policies

Cache is limited in size and capacity. So if there is a cache miss, we need to insert an a new entry into the cache. However, if the cache is full, we will need to evict an older entry. Therefore we need a policy based on which we can evict an older entry. Two common policies are LRU and LFU.

LRU

LRU or least recently used policy basically evicts the least recently used entry. This policy is fairly easy to implement and gives good results. LRU works well because same queries will generally occur close together with respect to time.

LFU

LRU or least frequently used policy basically evicts the least frequently used entry. This policy generally doesn’t give good results. LFU works well in some systems like CDN (content delivery network) where the frequency of queries are generally high and the pattern of queries don’t change much with time.

In practice, policies combine one or more of these based on the needs of the application. More information can be found here [1].

7.2 Cache Location

The location of the cache in the system is a very crucial design decision. It determines the performance, consistency guarantees and application complexity. Often caches are present at multiple locations in the systems.

Database Cache

This cache is close to the database or in the database itself. This generally doesn't have much performance benefit but provides great consistency.

Application Cache

This cache is close to the application server. This generally has great performance benefit as the cache is very close to the server. However, consistency becomes an issue as it is difficult to synchronize multiple application caches. Also the total cache memory may not be well utilized as application caches are local to the server and not shared between servers. This can lead to excessive cache misses if not designed correctly.

Global Cache

This cache is an independent system in itself which sits in between the database and the application server. It can be distributed over several nodes but can be abstracted as a single global cache. This provides good performance benefit, guarantees eventual consistency and can be scaled independently of the database and application server. Some popular global caches are Redis and Memcached.

7.3 Cache Write Policies

One of the biggest concerns with using a cache is cache invalidation during updates. Different systems need different needs. Generally, a trade off has to be done with respect to consistency and performance. For example, systems like CDN(content delivery network) will trade off performance over consistency. The following are 3 common write policies.

Write-through cache

Under this policy, data is written into the cache and the corresponding database at the same time. The cached data allows for fast retrieval and, since the same data

gets written in the permanent storage, we will have complete data consistency between the cache and the storage. Also, this scheme ensures that nothing will get lost in case of a crash, power failure, or other system disruptions.

Although, write through minimizes the risk of data loss, since every write operation must be done twice before returning success to the client, this policy has the disadvantage of higher latency for write operations.

Write-around cache

This technique is similar to write through cache, but data is written directly to permanent storage, bypassing the cache. This can reduce the cache being flooded with write operations that will not subsequently be re-read, but has the disadvantage that a read request for recently written data will create a “cache miss” and must be read from slower back-end storage and experience higher latency.

Write-back cache

Under this policy, data is written to cache alone and completion is immediately confirmed to the client. The write to the permanent storage is done after specified intervals or under certain conditions. This results in low latency and high throughput for write-intensive applications, however, this speed comes with the risk of data loss in case of a crash or other adverse event because the only copy of the written data is in the cache.

8 Time and Ordering in Distributed Systems

In a distributed system like super computers, the network is synchronous which implies that there is a common global clock. However, in most commercial distributed systems the network is asynchronous and each node has a local clock. This leads to a problem of ordering events in a distributed system where since each node has a local clock, the local timestamps cannot be used to order events.

Reference: Cloud Computing Concepts 1 [4]

One way of solving the issue is to synchronize the clocks. There are some difficulties associated with this.

- Each process runs on their own clock, we cannot assign one clock as the correct clock
- There is clock drift (difference in speed) and clock skew (difference in time)
- There are no bounds on how much delay is there in message transfer

- There is also no bound on time on processes running individual instructions

Popular protocols for clock synchronization are Christian's algorithm and NTP (Network Time Protocol). These protocols are susceptible to errors due to RTT and other delays.

In practice, we need to only order casually related events, that is if 2 events are causually related, we need to be able to compare them.

8.1 Lamport Clocks

Lamport Clock solution is to order events logically. Basically if event A happens before event B then timestamp(A) < timestamp(B).

The rules are as follows.

Note: Happens before are denoted by →, so A before B is denoted by A → B.

Rules

- A → B, timestamp(A) < timestamp(B)
- send(m) → receive(m)
- A → B and B → C then A → C

Algorithm

- Each process maintains a local counter which is an integer with initial value 0.
- A process increments the counter when a send event or any other event happens as it.
- Initial value is 0. The counter value is assigned as the timestamp of that event.
- The send event carries that timestamp alongside the message.
- For a receive event, the counter is updated by max(localCounter, message-Timestamp) + 1.

While lamport clocks allows us to compare causually related events. It has one issue which is two events with different timestamps needn't be causually related. Basically it cannot distinguish concurrent events. This issue is addressed by vector clocks.

8.2 Vector Clocks

Vector clocks build on Lamport systems by maintaining a vector of integer clocks. Each process maintains an the integer clock separately.

The j^{th} element of the vector clock at process i, $V_i[j]$ is i's knowledge of the latest events at process j.

Updating Vector Clocks

When an event happens at process i, only the i^{th} element of V_i is incremented. Each message sent between processes is accompanies by the sender's vector clock at the time of sending the message.

If process i receives a message, it first increments $V_i[i]$. Then it performs a vector max operation as follows.

$V_i[j] = \max(V_i[j], V_{message}[j]), \forall j$

Casually related events and concurrent events

Consider two events with vector clocks VT_1 and VT_2. We can determine if they are casually related or concurrent as follows.

$VT_1 = VT_2 \iff VT_1[i] = VT_2[i], \forall i \in [1..N]$

$VT_1 \leq VT_2 \iff VT_1[i] \leq VT_2[i], \forall i \in [1..N]$

Two events are casually related if $VT_1 < VT_2$ which is

$VT_1 \leq VT_2$ and $\exists j, VT_1[j] < VT_2[j]$

Two events are concurrent if NOT($VT_1 \leq VT_2$) and NOT($VT_1 \leq VT_2$).

9 Consensus Problem

The consensus problem states that there are N processes and each process has an initial variable value. They should reach consensus and give an output variable. The output variable at the end should be the same for all processes. There can be additional constraints such as validity that is every process has the same initial value, that is the final value as well. Integrity states that the final value should be proposed by some process.

Reference: Paxos Made Simple Paper [18].

The consensus problem is an abstract problem which can solve various problems such as -

1. Processes receive updates in the same order (reliable multicast)
2. Maintaining perfect membership lists (failure detection)
3. Leader election - a group of processes elect a leader to perform some task.

4. Distributed lock - mutually exclusive access to a resource.

The famous FLP impossibility proof states that consensus in an asynchronous distributed system is not possible. However, the paxos solution to the consensus problem can guarantee eventual consensus (eventual liveliness).

I will summarize the gist of the algorithm here, for full details, please check the reference.

Paxos

The paxos solution by Lamport Leslie guarantees two things.

Safety - Consensus is not violated - 2 processes do not decide on different value

Eventual Liveliness - Eventually the system will converge and works, that is the system will eventually decide value.

Paxos Rounds

- In the Paxos solution, there will be rounds. The rounds are asynchronous.
- A new round is started if the previous round doesn't result in consensus.
- If a process is in round j and it hears a message from round j + 1, it will abandon that round and move to the next.
- Processes can move to the next round based on a timeout.

Each round consists of 3 phases - the election phase, the bill phase and the law phase.

Election Phase

- Each process is a potential leader and chooses a unique ballet id. Uniqueness can be guaranteed by including process ID in the ballet id.
- Each process sends its ballet id to all processes.
- Processes wait for a while and respond to the highest ballet id. If a potential leader sees a higher ballet ID, it cannot be the leader.
- If in the previous round a value v' was decided, it includes this value in the response.
- If a potential leader receives a quorom of OKs, it is the leader.

A key point in the paxos algorithm is that once a process decides a value, it is not changed. This guarantees safety.

Bill Phase

- Leader sends proposed value to all, if a value was decided in the previous round
- Recipients log the value on to the disk and respond OKs.

Law Phase

- If the leader receives a majority of OKs, it lets everyone know of the decision.
- Recipients receive the decision and log it on the disk.

The Paxos algorithm is used in a variety of applications including Zookeeper which is a distributed synchronization service and Chubby which is a distributed lock service. Readers should note that I have just summarized the gist of the paxos solution and should refer the paper for more details.

10 MySQL

The MySQL system be design is traditionally a single server monolithic database. Nevertheless, there are several ways the database can be distributed within certain limitations. Lets look into the built in database engines.

10.1 Storage Engines

MySQL has been designed to make it relatively easy to add other storage engines. This is useful if you want to provide an SQL interface for an in-house database.

Reference: [12]

1. **InnoDB** - This is the most popular engine. It supports transactions and is ACID compliant. It allows row level locking, crash recover and is the only engine to support foreign key integrity referential constraint.
2. **MyISAM** - This engine is very fast but it doesn't support transactions and onlt supports table level locking. It is generally used in data warehousing.
3. **Memory** - This is the fastest engine as everything is stored in memory. There are no transactions and everything is lost when the database is restarted.
4. **CSV** - The data is stored in a CSV file, which is good for portability and integrating with other applications.

10.2 Data Replication

MySQL supports Master Slave Architecture. Replication allows the master MySQL server to be replicated asynchronously over slave servers. Depending on the configiguration, either entire databases or single tables can be replicated.

Replication has the following advantages

1. High read throughput
2. Data redundancy
3. Analytics, Backups etc which are heavy tasks can happen on a slave DB
4. Long distance data distribution

10.3 Partitioning

Partitioning allows to distribute portions of a table over a file system.

Some of the hash functions are RANGE, LIST, and [LINEAR] HASH partitioning. The value of the partitioning column is passed to the partitioning function,

which returns an integer value representing the number of the partition in which that particular record should be stored. This function must be non constant and nonrandom.

MySQL only supports horizontal partioning.

The primary advantage of partitioning is that more data can be stored and the read and write throughput is greatly improved. Sharding is one specific type of partitioning, namely horizontal partitioning.

10.4 Distributed MySQL Servers

The default MySQL edition doesn't natively support configuring it as a distributed database. However, MySQL server can still be distributed at the application level by sharding the database and using consistent hashing. However, this poses several limitations as previously discussed. An alternative is to use a MySQL variant such as MySQL Cluster CGE. I was unable to find much information on the same.

11 Cassandra

Cassandra is a columnar NoSQL database. Cassandra is a peer-to-peer distributed database that runs on a cluster of homogeneous nodes. It is a highly available database that favours availability and partition tolerance over consistency. It follows BASE properties. The consistency level is tunable by setting the quorom level.

Cassandra follows a shared nothing architecture. A Cassandra cluster has no special nodes i.e. the cluster has no masters, no slaves or elected leaders. This enables cassandra to be highly available with no single point of failure.

All images and some content referenced from [11]

11.1 Datamodel

The query language supported by Cassandra is like SQL but the datamodel is completely different. Analogous to tables in SQL, Cassandra has a column family. A record is identified by row key and the column key.

In Cassandra, writes are cheap in comparison to joins, group by and aggregation. So read performance is optimized by increase the number of data writes (in order to minimize reads). De-normalization is encouraged as disk space is cheap as compared to memory and CPU processing and IO operations. For example, if a join is frequently needed, create a table with the joined data. Another example, if a table needs to be frequently indexed on 2 columns then create 2 separate tables each with a different index.

11.2 Architecture

In Cassandra data is partitioned horizontally over shards and the shards are arranged around a ring and consistent hashing is used. Data is replicated over multiple nodes according to the replication factor ensuring that the data is always available in the event of failures.

The client sends a request to any node, that node becomes the coordinator. The node can then send the query to the relevant nodes (as described by consistent hashing) and get the result based on quorum. Cassandra follows last write wins policy. Quorom will be discussed later.

A cassandra node has 3 primary data entities -

1. Mem Table - This is the in memory store which is very fast but volatile. Recent writes are present here before being flushed to store.
2. Commit Log - this is used to ensure durability

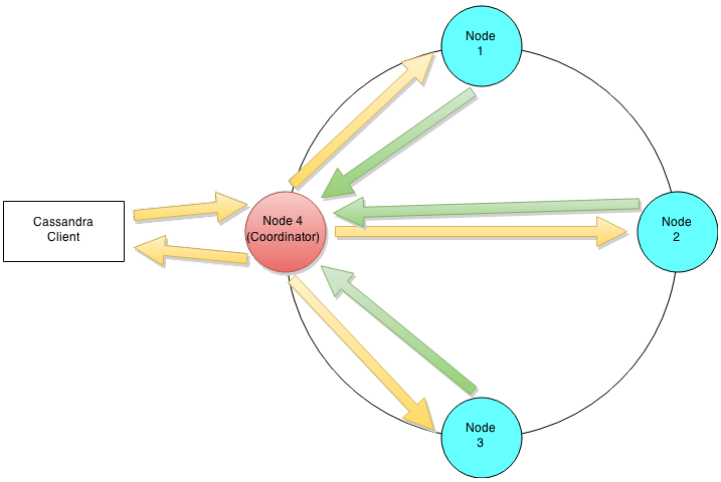

Figure 11.1: Cassandra client request

3. Set of SSTables - Cassandra maintains a set of sorted string tables which are immutable.

11.3 Write Path

Each node associated with the data on the ring receives a write request. We will discuss the individual write path.

- First, an insertion is made into the log file. This ensure durability of the database.
- An entry is made into the memtable.
- If the update is deletion, then a tombstone entry is made into the memtable
- The write node responds with an OK acknowledgement.

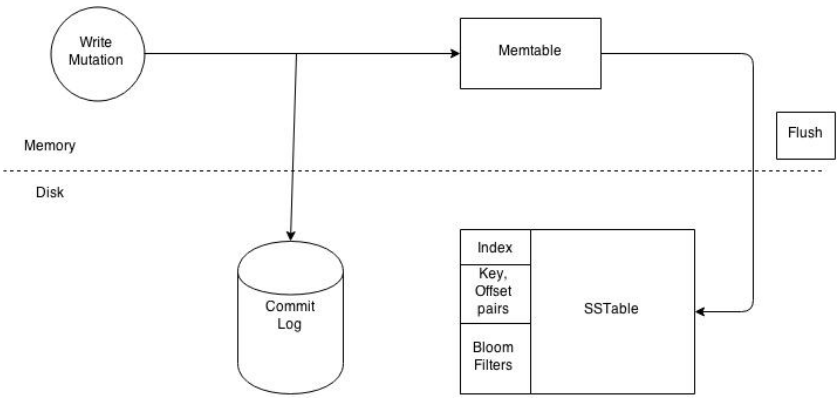

Figure 11.2: Cassandra write path

If the node crashes then the memtable can be recreated from the commit log. Once the memtable is full or the user requests we can flush the memtable into disk as a SSTable. Each SSTable is associated with a bloom filter, index file and datafile. The datafile is sorted on the index. Bloom filter and index file is used to speed up read queries as discussed in the next section. SSTables need to be regularly combined to save space, this is known as compaction.

Since each write request just needs an entry in the commit log and the memtable, writes are incredibly fast.

11.4 Read Path

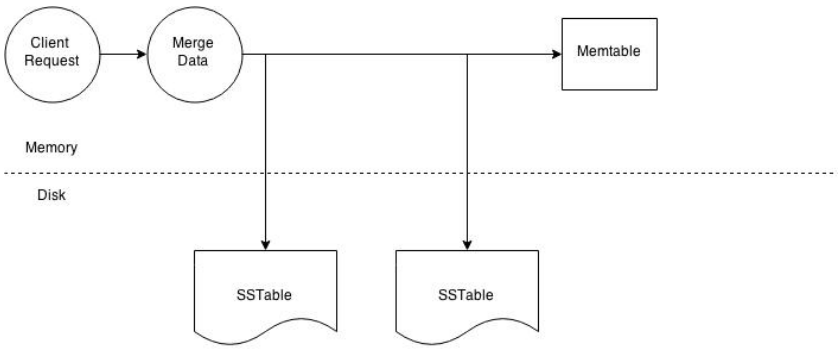

Figure 11.3: Cassandra read path

Individual node read operations happens as follows. First the key is looked up in the mem table, if present it returns the query from there. Otherwise it starts looking for it in the individual SSTables. Each SSTable has a bloom filter helps in quickly testing if the record is present or not in the SSTable. The bloom filter test can give false positives but can never give false negatives. The Bloom filter can be stored in memory and therefore is fast.

As we can observe, the read path is much more complex than the write path. So in practice, schemas are heavily denormalized to make the read path efficient.

11.5 Coordinator and Quorom

If N is the replication factor, the coordinator needs to send the read or write request to all N replicas. However, the coordinator has to wait for only R replicas to respond for a read request and only W replicas to respond for a write request. If consistency is desired, then R + W > N. Generally N is 3 and R and W are 2. In the read path, if any of the replicas are outdated, a read repair is sent to them. By pigeon hole principle, quorom achieves consistency.

12 Google - Big Table Database

Bigtable [15] is a distributed storage system developed by Google for managing structured data at a large scale. It provides clients with a simple data model that supports dynamic control over data layout and format. The bigtable paper is one of the most influential papers in distributed computing and storage.

12.1 Datamodel

Bigtable is a sparse, distributed, persistent multidimensional sorted map. Each record is identified by a row key, column key and a timestamp. Read and writes of data in a single row key is atomic. Bigtable doesn't support atomic transactions over multiple rows.

Columns are grouped together into column families which help in locality and compression as, data of column families are compressed together. Each column is identified by *family:qualifier*.

Rows are stored in sorted order of row key in the database. The row range for a table is partitioned and each row range is called a tablet. This forms the unit of distribution and load balancing.

12.2 Architecture

Bigtable builds on several building blocks include Google File System [17] and Chubby [14]. GFS is a distributed file system while chubby is a distributed lock service.

The Google SSTable file format is used internally to store Bigtable data. This is similar to the Cassandra SSTable.

The Chubby service consists of five active replicas, one of which is elected to be the master and actively serve requests. Chubby is used to acquire locks on directories and files.

There are many uses of Chubby

1. Ensure that there is only one master at any time (single leader)
2. Store the root tablet of bigtable data
3. Discover and finalize tablet server deaths
4. Storing the bigtable schema information

If chubby fails for an extended period of time, bigtable also becomes unavailable. We note here that bigtable is favouring consistency over availability by using a centralized architecture. We will see that this usually isn't much of a issue as

the master generally isn't heavily loaded and clients rarely communicate with the master.

Apart from the master, big table consists 2 other major components - tablet servers and the client library.

12.3 Master

The master is responsible for several book keeping tasks. Clients rarely communicate with the master. Some of the responsibilities of the masters are as follows

1. Assigning tablets to tablet servers based on server load
2. Detecting that tablet servers are unavailable and reassign tablets as soon as possible. This is done by the master repeatedly asking the tablet server the status of its lock on the tablet. If the master cannot reach the tablet server then the master attempts to acquire an exclusive lock on the tablet file.
3. The master itself acquires a unique master lock in chubby to avoid multiple instances of a master server.
4. The master is also responsible for garbage collection.

12.4 Tablet location

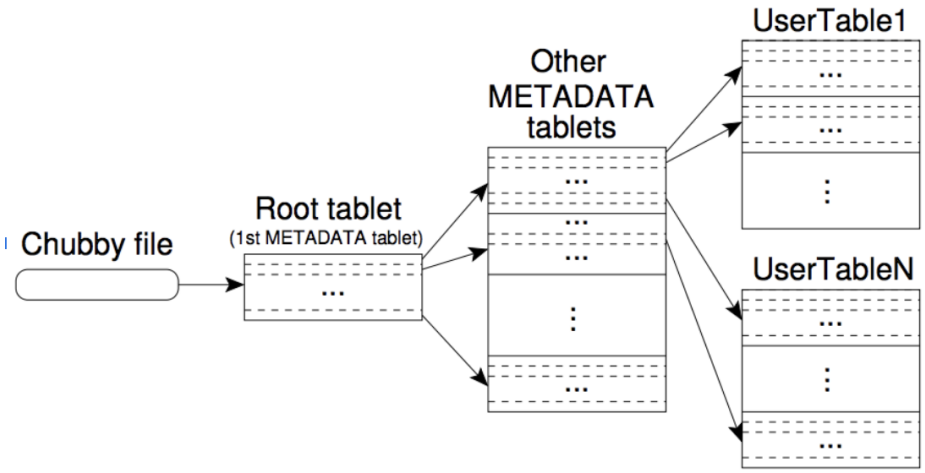

Figure 12.1: Bigtable - tablet locations

The first level is a file stored in Chubby that contains the location of the root tablet. The root tablet contains the location of all tablets in a special METADATA table. Each METADATA tablet contains the location of a set of user tablets.

The root tablet is just the first tablet in the METADATA table, but is treated specially—it is never split.

The METADATA table stores the location of a tablet under a row key that is an encoding of the tablet's table identifier and its end row.

The client library caches tablet locations. If the client does not know the location of a tablet, or if it discovers that cached location information is incorrect, then it recursively moves up the tablet location hierarchy.

12.5 Tablet Servers

Each tablet is assigned to one tablet server at a time. This ensures consistency. The actual persistent state of the tablet is stored in the GFS. Recent mutations are stored in a commit log and an in memory table. Older updates are stored in SSTables. This is very similar to what Cassandra does.

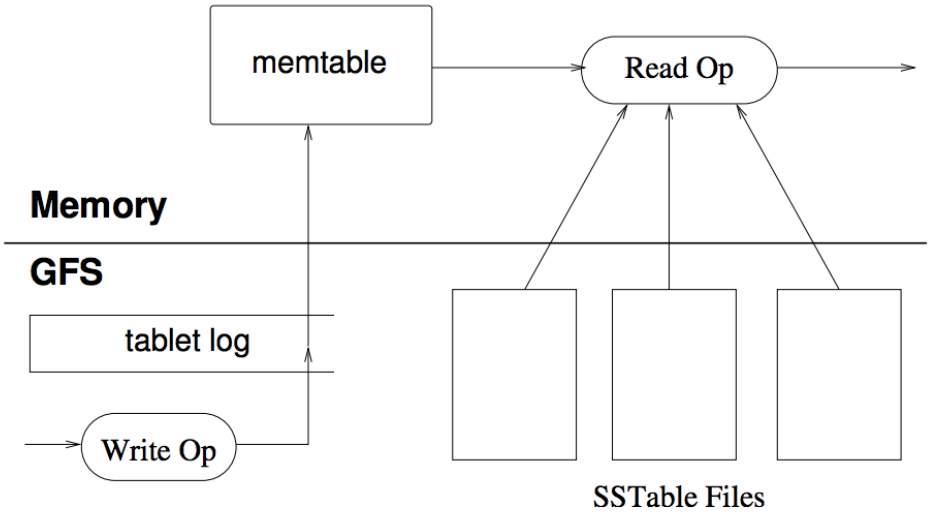

Figure 12.2: Bigtable - tablet serving

12.6 Refinements

The paper talks about many refinements to achieve performance and scale.

- **Locality groups** - This allows column families to be grouped together to allow more efficient reads.
- **Compression** - Combining compression with locality groups, SSTables can be highly compressed

- **Bloom Filters** - These filters help in quickly testing if a key is present in an SSTable.

The reader is suggested to read the paper for more information.

13 Amazon - Dynamo Database

At Amazon, the servers receive a lot of traffic at peak times. Reliability is essential and even small down times can have significant consequences in customer trust and brand. Amazon servers are implemented on infrastructure which span multiple data centers across the world and components fail all the time.

The Dynamo paper [16] presents the design and implementation of Dynamo, a highly available hey-value storage system. This database provides an always on experience that never rejects writes even during significant outages. It resolves conflicts by using object versioning and application assisted conflict resolution that allows semantic resolution. Dynamo provides a novel interface which is one of the first of its kind.

In Amazon's platform, services have very strong latency requirements. They are generally measured at the 99.9th percentile of the distribution. That is they don't take median or average as they want to ensure good performance to all clients. The choice of 99.9 was done after a cost-benefit analysis which indicated a significant increase in cost to improve performance after that. The operations of Dynamo is considered to be non-hostile and there are no security related requirements as those are handled at the application layer.

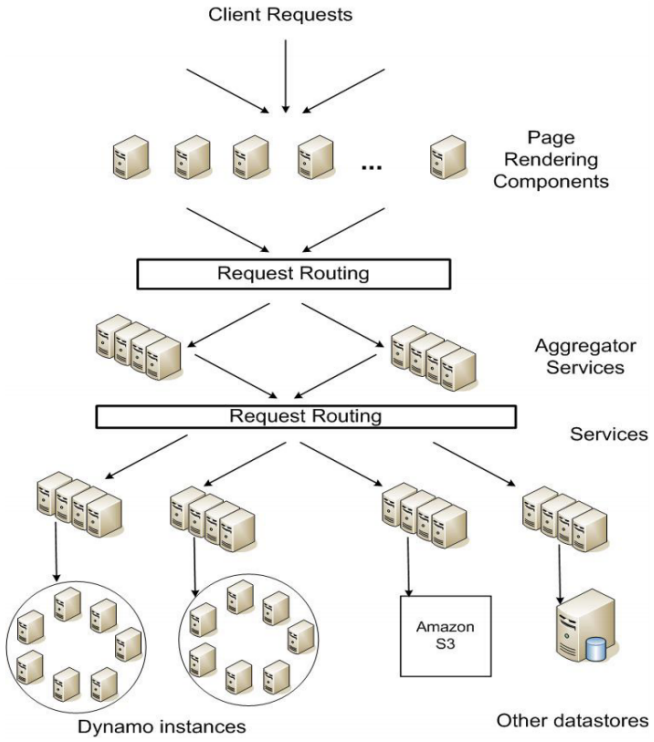

Figure 13.1: Dynamo - Service Oriented Architecture

13.1 Datamodel

Most of the services on Amazon's platform require only a primary-key access to the datastore. So Dynamo supports simple read and write operations to a data item that is uniquely identified by a primary key. The value can be anything. There are no operations that span multiple data items and hence there is no need for a relational schema.

13.2 Datastore Design

In order to ensure availability in a system prone to server and network failures, availability is increased by trading off consistency for eventual consistency. The challenge with this approach is that it can lead to conflicting changes which must be detected and resolved. This conflict resolution poses two questions, who should resolve them and when.

Many traditional data stores execute conflict resolution during writes and keep the read complexity simple. But Dynamo wants to be an "always writable" data store so conflict resolution has to be pushed into the read path to allow writes under partition.

Either the datastore or the application can resolve the conflict. However, the datastore can only have a simple policy like last write wins. On the other hand the application can handle conflict resolution using semantics of the data.

Key Principles

The paper cites a few key principles in Dynamo.

1. **Incremental scalability** - Dynamo should be able to scale easily and linearly with the requirements.
2. **Symmetry** - Each node should have the same set of responsibilities as its peers. That is there is no master-slave architecture or design. This is a key requirement to ensure availability.
3. **Decentralization** - An extension to symmetry, the design should favour peer to peer control over a centralized control.
4. **Heterogeneity** - Dynamo needs to be able to exploit the difference in the hardware capabilities of individual nodes in the infrastructure it runs.

13.3 Architecture

The techniques used to solve the various problems in Dynamo are shown at a glance in figure 13.2. The following sections will discuss some of these in detail.

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Figure 13.2: Different Techniques used in Dynamo

13.3.1 Query Interface

In Dynamo, there are only two operations get() and put(). Each object is uniquely identified by a key. However, objects associated with a single key are also identified by a context to differentiate the various version of the same object. The context basically consists of a vector clock and will be discussed in a later section.

The put(key, context, object) operation determines where the replicas of the object should be placed based on the associated key, and writes the replicas to disk. The context encodes system metadata and a vector clock about the object that is used in conflict resolution.

The get(key) operation locates the object replicas associated with the key in the storage system and returns a single object or a list of objects with conflicting versions along with a context.

13.3.2 Partitioning and Replication

Dynamo uses consistent hashing with virtual nodes to partition the datastore. Each data item is replicated at N nodes, where N is configurable. Each key, k is assigned to a coordinator node in the ring. The coordinator is responsible for replicating the data on the next N-1 clockwise successor nodes. The list of the nodes are called preference list.

13.3.3 Data Versioning

This is the most critical part of the dynamo architecture. In order to be an always writable system, the system should support multiple divergent versions of data. Dynamo creates each modification as a new and immutable version of the data. Each object version is associated with a vector clock as illustrated in figure 13.3.

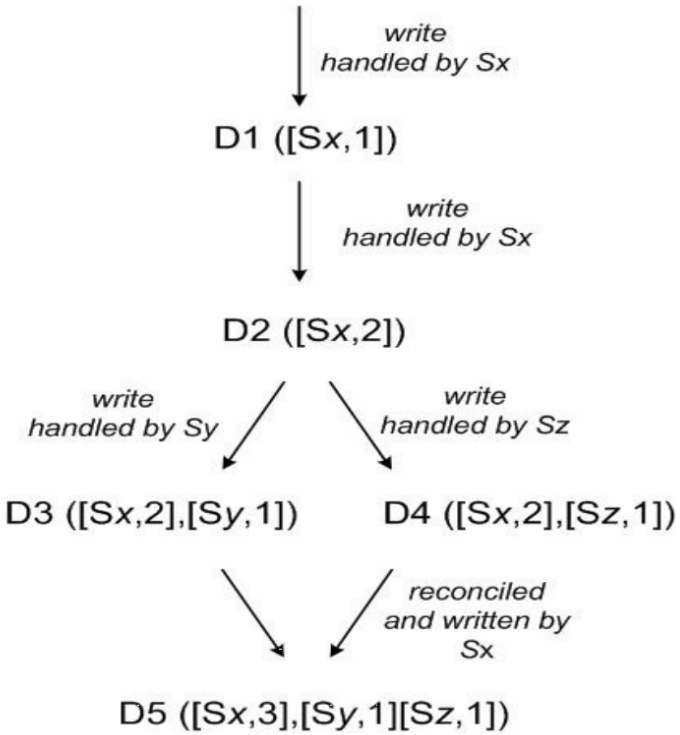

Figure 13.3: Object Versioning Example

Syntactic Reconciliation - Most of the time, new versions of the data overwrites previous versions. The system itself can determine the authoritative version. This is done by checking if there is a causal ordering between the two version's vector clocks. During normal execution without multiple concurrent node failures or prolonged network partition, syntactic reconciliation takes place.

Semantic Reconciliation - In the presence of multiple failures combined with concurrent updates, version branching will take place. Here the system cannot determine the authoritative version as there is no casual ordering between the different versions. Here the application should apply semantic based conflict resolution.

In Dynamo when a client issues a read/get request it gets a response with all the different versions and needs to resolve the conflict at the same time. Similarily, when a client wants to update an object, it must specify which version it is updating.

A possible issue with vector clocks is that the size of vector clocks may grow if many servers coordinate the writes to an object. In practice, this is not likely because the writes are usually handled by one of the top N nodes in the preference list.

13.3.4 Handling failures - Hinted Handoff

As dynamo wants to be an always write able store, during network partition, it cannot wait for quorom to allows writes. To resolve this, dynamo doesn't enforce strict quorom membership and instead uses a sloppy quorom. All read and writes are performed on the first N healthy nodes from the preference list which may not be the first N nodes on the consistent hashing ring. This is known as hinted handoff.

Nodes that receive hinted replicas will keep them in a separate local database that is scanned periodically. Upon detecting the target nodes have recovered, the node will attempt to deliver the replica to the target nodes. Once the transfer succeeds, the node may delete the object from its local store without decreasing the total number of replicas in the system.

Readers should note that Hinted handoff works best if the system membership churn is low and node failures are transient.

13.4 Conclusion

We discussed the Dynamo always writable architecture in detail. We saw how Dynamo handles conflict resolution when divergent versions of the same object are created during partition. We also discussed hinted hand off and sloppy quorom which ensure writes always succeed during partition.

14 Google File System

The Google File System [17] is a scalable distributed file system for large data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware. The system was needed to meet the storage needs at Google for various services. Several design decisions were made based on the usage patters on the services.

14.1 Design Observations

Based on the needs of several services, some of the key observations for the design were

1. Failures are the norm rather than the exception. The system should support multiple arbitrary failures
2. Files are large and can run into several gigabytes and the system should optimize for them. The system should support small files as well but need not optimize for them.
3. Files are generally mutated by appending to them rather than random writes in the system. Random writes need to be supported but are discouraged.
4. Once files are written, they are generally read sequentially. Throughput is more important than latency. That is the worlload generally consists of large streaming reads and small random reads.
5. The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file.

14.2 Architecture

The GFS cluster consists of a single master and multiple chunkservers. The chunkservers store the chunks as regular linux files. The chunk servers are accessed by multiple clients. Files are divided into fixed size chunks (64 MB) and these chunks are distributed over the chunk servers. Each chunk is identified by a unique 64 bit handle and is assigned by the master.

Master Server

Some of the responsibilities of the master are

1. Creation and management of file name space

2. Mapping from file to file chunk identifiers - Creation and management of file chunk identifiers
3. The master alls maintains all file system metadata. This includes the access control information and the current locations of chunks.
4. It also controls system-wide activities such as chunklease management, garbage collection of orphaned chunks, and chunk migration between chunk servers.
5. The master is also responsible to detect chunk server failures which it does so by periodic heartbeat messages. It also is responsible for assigning of chunks to new chunk servers.

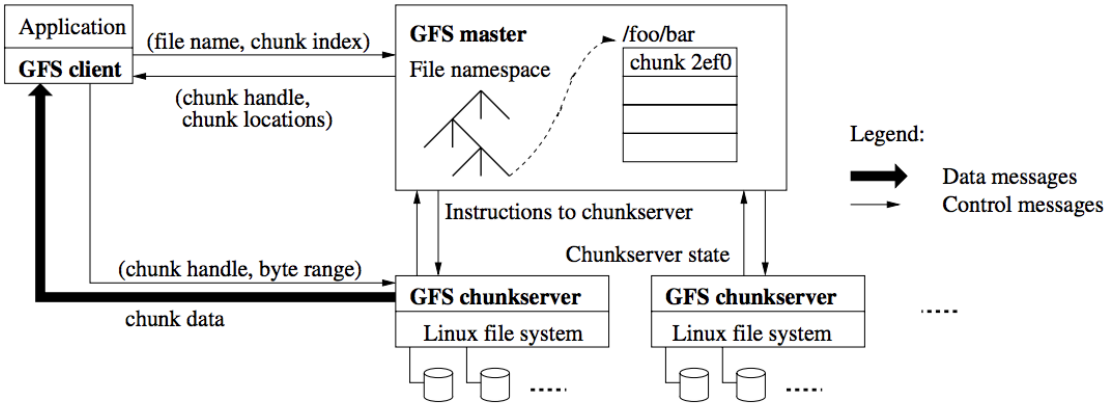

Figure 14.1: Google File System Architecture

Having a single master vastly simplifies the design and enables the master to make sophisticated chunk placement and replication decisions using global knowledge. However, the architecture must minimize its involvement in reads and writes so that it does not become a bottleneck. Clients never read and write file data through the master. Instead, a client asks the master which chunkservers it should contact. It caches this information for a limited time and interacts with the chunkservers directly for many subsequent operations. All data operations communication directly goes to the chunkservers.

Chunks are 64 MB which is greater than the typical block size. Lazy space allocation avoids wasting space due to internal fragmentation.

The master persists the file and chunk namespaces and the mapping from files to chunks. However, it doesn't persist the chunk replica locations and instead it asks each chunkserver for their chunks on startup. This is an important design decision because the chunk servers may fail, get corrupted etc in a system failure.

The operation log at the master contains a historical record of critical metadata changes. Not only is it the only persistent record of metadata, but it also serves as a logical time line that defines the order of concurrent operations. The master

recovers its file system state by replaying the operation log. Checkpoints are made to optimize the process.

14.3 Consistency Model

File namespace mutations (e.g., file creation) are atomic. They are handled exclusively by the master. When a file mutation happens without interference from concurrent writers, the write is considered defined and consistent. On the other hand when there are consistent concurrent successful mutations, it leaves the region undefined but consistent. Basically the fragments from concurrent mutations are mixed.

A record append causes data (the "record") to be appended atomically at least once even in the presence of concurrent mutations, but at an offset of GFS's choosing.

The relaxed consistency model means applications have to take accommodate for inconsistencies using some techniques such as checkpointing.

14.4 Leases and Mutation Order

GFS uses leases to ensure consistent mutation order across replicas. The master grants one of the chunk replicas a lease. This chunk is now the primary. The lease expired in 60 seconds. The primary picks a serial order for all mutations for the chunk. All the other replicas follow this order while applying mutations hence guarantying consistency in order.

The lease mechanism is designed to minimize management overhead at the master.

The steps involved for mutations are as follows

1. The client asks the master which chunkserver holds the current lease for the chunk. The master assigns a lease if not present.
2. Master replies with primary and secondary chunks
3. Client sends data to all replicas
4. Once all replicas have received the data, it sends a write to the primary specifying the data identifier
5. The request identifies the data pushed earlier to all of the replicas. The primary assigns consecutive serial numbers to all the mutations it receives, which provides the necessary serialization.
6. The primary forwards the write request to all secondary replicas. Each secondary replica applies mutations in the same serial number order assigned

by the primary.

7. The secondaries all reply to the primary indicating that they have completed the operation.

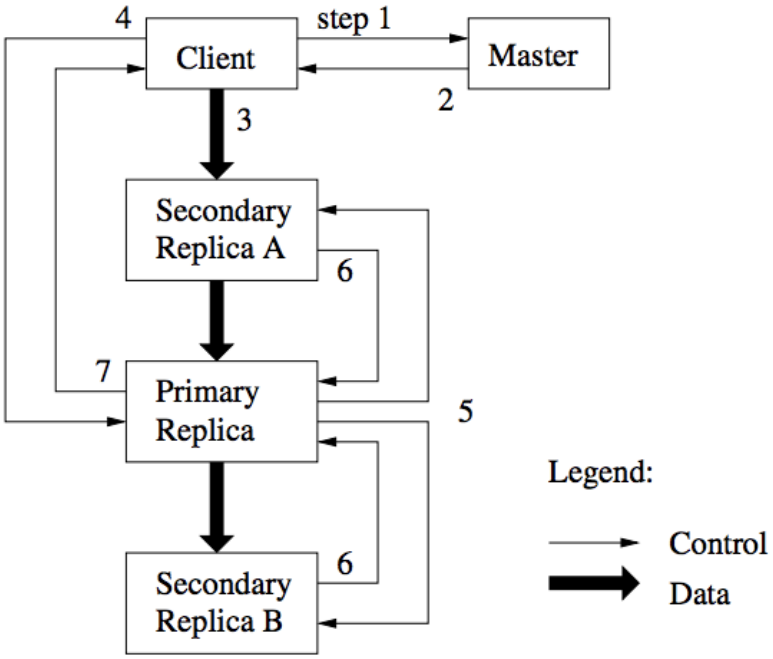

Figure 14.2: Write Control and Data Flow

If a write by the application is large or straddles a chunk boundary, GFS client code breaks it down into multiple write operations

14.5 Data Flow and Atomic Appends

We decouple the flow of data from the flow of control to use the network efficiently. While control flows from the client to the primary and then to all secondaries, data is pushed linearly along a carefully picked chain of chunkservers in a pipelined fashion.

Atomic Record Appends

1. The client pushes the data to all replicas of the last chunk of the file.
2. Then, it sends its request to the primary.
3. The primary checks to see if appending the record to the current chunk would cause the chunk to exceed the maximum size (64 MB).

4. If so, it pads the chunk to the maximum size, tells secondaries to do the same, and replies to the client indicating that the operation should be retried on the next chunk.
5. If the record fits within the maximum size, which is the common case, the primary appends the data to its replica, tells the secondaries to write the data at the exact offset where it has, and finally replies success to the client.

14.6 Snapshot

The snapshot operations allows copying of a file or a directory instantly wile minimizing and ongoing mutations.

Users can use it to quickly branch copies of files or to checkpoint state before experimenting with changes. It is a sort of version control.

GFS uses standard copy-on-write techniques to implement snapshots. After the leases have been revoked or have expired, the master logs the operation to disk. It then applies this log record to its in-memory state by duplicating the metadata for the source file or directory tree. The newly created snapshot files point to the same chunks as the source files.

14.7 Conclusion

We discussed the google file system architecture. We saw that one of the important design decisions was to support efficient appends rather than random writes. The architecture contains a single master and many slave servers. The master does some book keeping tasks and maintains the file namespaces and other metadata. This vastly simplifies the design and architecture. We saw how the dataflow and control path are separated and how that makes things faster but consistent. We also discussed how simultaneous atomic records appends are performed. Finally we discussed snapshots which is used for versioning and backups which uses COW technique.

15 Reading Elective - Conclusion

15.1 Summary

We have discussed different algorithms and techniques for creating distributed storage. We have also discussed different case studies which leverage the concepts. We can now therefore make some abstractions.

Communication between processes

We have many techniques to handle communication between processes prone to failure and network partition. Gossip protocol can handle efficient and robust message passing. Membership lists allow us to know the different processes in the system. Merkle trees allow us to quickly find difference in large blocks of data and allow efficient transfer. The Paxos algorithm can be used if different processes in the system need to decide on something. These various techniques can be combined to handle communication between processes.

Consistency

In many systems, it is the norm to trade strict consistency for eventual consistency in favour of availability and partition tolerance. Eventual consistency can be guaranteed by quorum systems. Vector clocks help in ordering events on different processes. Further, Vector clocks combined with syntactic and semantic reconciliation can resolve conflicts in prolonged network partition.

If strict consistency is required, it is very difficult to guarantee in a peer to peer system. A master process is generally needed but imposes a possible bottleneck and a single point of failure. Bottlenecks can be avoided by having the master do only light-weight book keeping and other control coordination tasks. A single point of failure cannot be entirely avoided, however the issue can be mitigated by having clients not interact directly with master allowing some room for a new master process to come in place.

Performance

In commercial systems, clients expect stringent response times. For data stores, a standard technique for distributing the load is data sharding combined with consistent hashing. Another technique useful for performance increase is caching. However, it is important to note that all these techniques come with decrease in consistency and this has to be handled separately.

Persistence of data

Processes fail all the time. In most cases a process failure indicates a crash in which main memory is lost but secondary memory is retained. In these cases, having logs for events and checkpoints allow quick recovery of processes. To mitigate the issue of complete disk failures, the system can have multiple replicas in different data centers for storing the data durably. A new process can now copy data from the replicas. Merkle trees can be used to make the process efficient.

15.2 Future Work

In this study, we looked at different algorithms, concepts and techniques for creating highly available distributed databases. We studied the architectures of 3 different types of NoSQL databases and one distributed file system. More case studies can be done on architectures which implement distributed consensus like zookeeper and chubby. Case studies on other distributed architectures like Cache(redis, memcached) and search (Elastic) can also be considered. While we now know how to store data in a distributed manner, there is lot of scope on studying the different algorithms and techniques in distributed compute. Some example frameworks are MapReduce and Spark. A parallel study can be done on how the different techniques and abstractions can be leveraged to build complete systems. Some other concepts to explore include load balancing, message queues etc. Lastly, complete case studies can be done on large scale systems such as Facebook and Twitter to study how the different architectures can be combined to produce one large system.

References

[1] Caching. https://en.wikipedia.org/wiki/Cache_replacement_policies.

[2] Cap theorem and distributed database management systems. https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e.

[3] Cloud computing concepts 1 - gossip protocol. https://www.coursera.org/learn/cloud-computing/lecture/5AOex/1-2-the-gossip-protocol.

[4] Cloud computing concepts 1 - vector clocks. https://www.coursera.org/learn/cloud-computing/lecture/7zWzq/2-1-introduction-and-basics.

[5] Consistent hashing. https://en.wikipedia.org/wiki/Consistent_hashing.

[6] Database sharding. https://www.digitalocean.com/community/tutorials/understanding-database-sharding.

[7] Distributed caching. https://www.youtube.com/watch?v=U3RkDLtS7uY&t=16s.

[8] Gossip and epidemic protocols. http://disi.unitn.it/~montreso/ds/papers/montresor17.pdf.

[9] Gossip protocol. https://en.wikipedia.org/wiki/Gossip_protocol/.

[10] An illustrated proof of the cap theorem. https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/.

[11] Introduction to apache cassandra's architecture. https://dzone.com/articles/introduction-apache-cassandras.

[12] Mysql - storage engines. http://zetcode.com/databases/mysqltutorial/storageengines/.

[13] M. Branco. Topology-aware gossip dissemination for large-scale datacenters, 2012.

[14] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[15] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: ama-

zon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. 2003.

[18] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[19] V. W.-H. Luk, A. K.-S. Wong, C.-T. Lea, and R. W. Ouyang. Rrg: redundancy reduced gossip protocol for real-time n-to-n dynamic group communication. *Journal of Internet Services and Applications*, 4(1):14, May 2013.