MIT 6.824 Distributed Systems

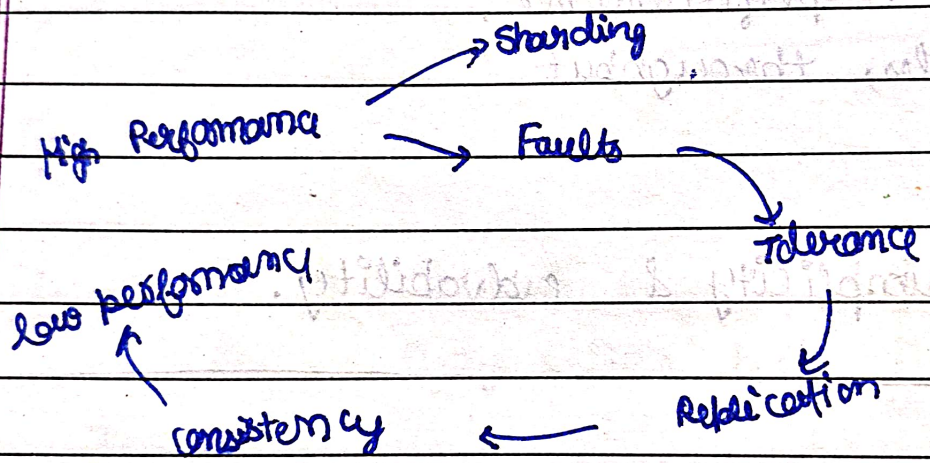
Infrastructure
- Storage
- Communication
- Computation

Course Contents
- RPC
- Threads
- Concurrency Control
- Performance
  - Scalability
  - Fault Tolerance
  - Consistency

⇒ RPCs & Threads.

Why threads?
- I/O concurrency
- Parallelism
- Convenience → ping, background tasks.

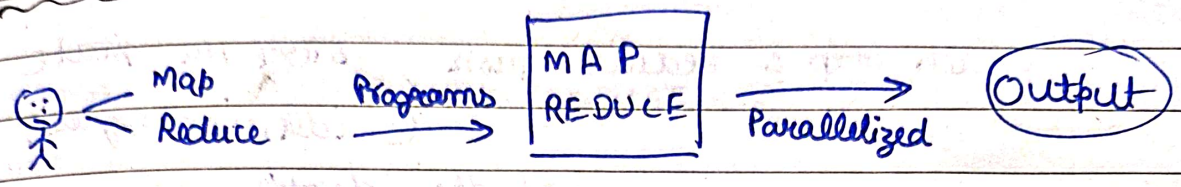
⇒ GFS



(2004)

MapReduce: Simplified Data Processing on Large Clusters

⇒ Abstract



- Partion Input-data
- Scheduling program's execution across a set of machines.
- Handles Machine failures.
- Manages Inter-Machine communication.
- High Performance

Input
↓ MAP
Set of Intermediate key/value pairs.
↓ REDUCE
Output which shares same intermediate key.

Hence,

Map (K1, V1) ⟶ list (K2, V2)

Reduce (K2, list(V2)) ⟶ list (V2)

e.g. Distributed GREP, count of URL access Frequency, Inverted Index, Distributed Sort, etc.

⇒ Implementation

Input Data —splits→ M sets —Processed in parallel→ Intermediate keys —splits→ R sets → Output

MAP

REDUCE

(Users can specify R partioning functions)

The implementation consists of the following:

(1) Masture Data Structure

For each Map & reduce task, store the state (idle, in-progress, or completed) and the identity of worker machines for non-idle tasks.

(2) Fault Tolerance

Worker (Just Reset).

Master (Can Make backup to manage the data structure)

(3) tries Locality, & Task Granularity:

⇒ Refinements

1. I/O types
2. Monitoring Service

Note: Mapreduce can solve problems that are trivial but big. Might be difficult for hard problems

i.e. batch processing jobs ✓

Interactive jobs ✗

The Google File System (2003)

⇒ Abstract

Scalable distributed file system over an inexpensive commodity hardware.

Traditional File Systems + Google's observations (extensions)
↓ distributed for their workload

(Performance + scalability + reliability + availability)

⇒ Introduction

Questioning existing design choices

- Component failures
  - They need to be considered as norm, since it is always fails in thousands of machines
  - Problems can occur by (system) OS bugs, apps bugs, human errors, disk fails, networking, power supplies.
  - Therefore, constant monitoring, errors detection, fault tolerance, & automatic recovery.

- Huge file sizes
  - moving billions of KB files are unwieldy, since multi GB files are common. I/O operation & block sizes have to be revisited.

- Write / Update considerations in files (Think of OASYS)
  - Most files are append-only, over-writing existing data are practically non-existent.
  - Once written, the files are only read, and often only sequentially.

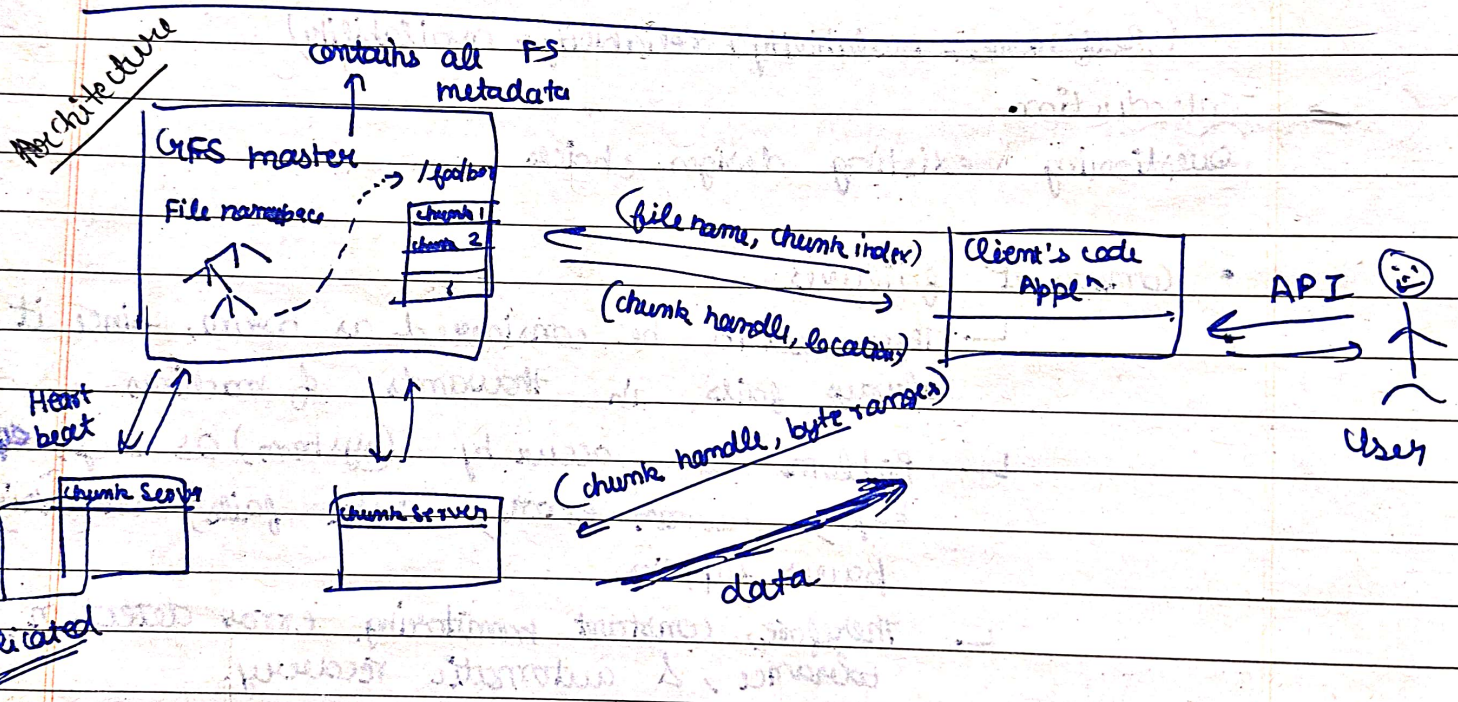
The workloads are often analytics, logs etc.

- Co-designing appl^n & the FS API
  - Adding atomic append operation

⇒ Design Overview

Operations Supported → Create, delete, open, close, read & write
+
Snapshot & record append

Architecture



FS metadata includes:
- Namespace
- Access control information
- Mapping from files to chunks
- Current location of chunks

+

Operations
- Chunk-lease mng.
- garbage dlection
- Chunk Migration

Forever
Page No.
Date : / /201

(2004)
Clusters

→ Single master
- Minimize its involvement in read & writes

→ Chunk Size
- 64 MB
- Reduces Metadata on the server, which allows to store it in memory.
- Reduces Request information.

→ Metadata
- All metadata is kept in the master's memory
- They are kept persistent by logging mutations to an operation log.

In-memory DS	Chunk locations	Checkpoints + Operation Log
1. Scanning periodically	1. Polls chunk-servers at startup for locations.	1. Record of critical metadata changes.
2. Fast & efficient operations	2. Can update using heart-beat messages.	2. Persistent & replicated to remote machines.
	3. Also by periodically asking.	3. Also, master creates a checkpoint whenever log grows beyond a certain size.

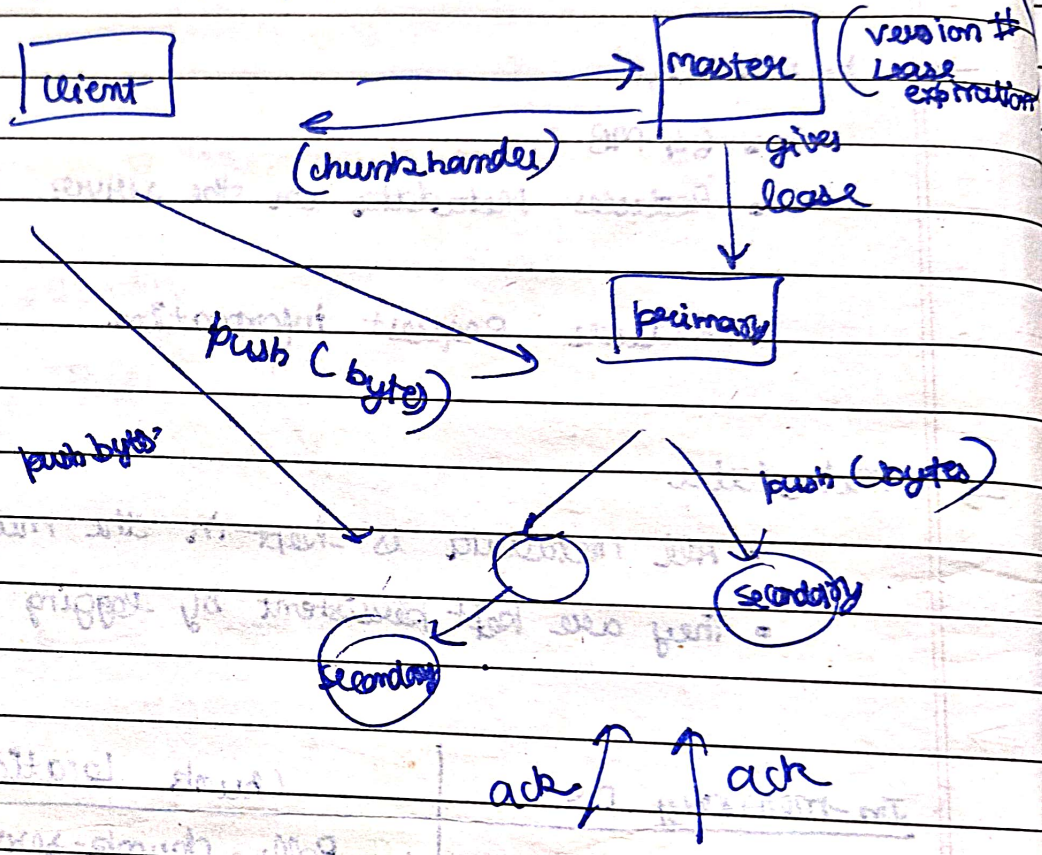
→ Consistency Model

Forever
Page No.
Date : / /201

⇒ Operations

Record appends

append (filename , bytes)



After ack, the client send write request & BOOM

after all "Yes", "success" → done

else → not done

* This still cause in-consistency,

some may append & some not.

VMWARE

Design of Practical System for Fault-Tolerant Machines (2010)

⇒ Abstract

(Works for fail-stop faults)
Fault Tolerant VMs via primary/backup replication approach.

Practical Issues + Design Choices & Alternates + Implementation.

⇒ Introduction

To replicate, we have two approaches:

i) State transfer: constantly send (all changes to the state of primary i.e. CPU, memory, I/O) to backup continuously.

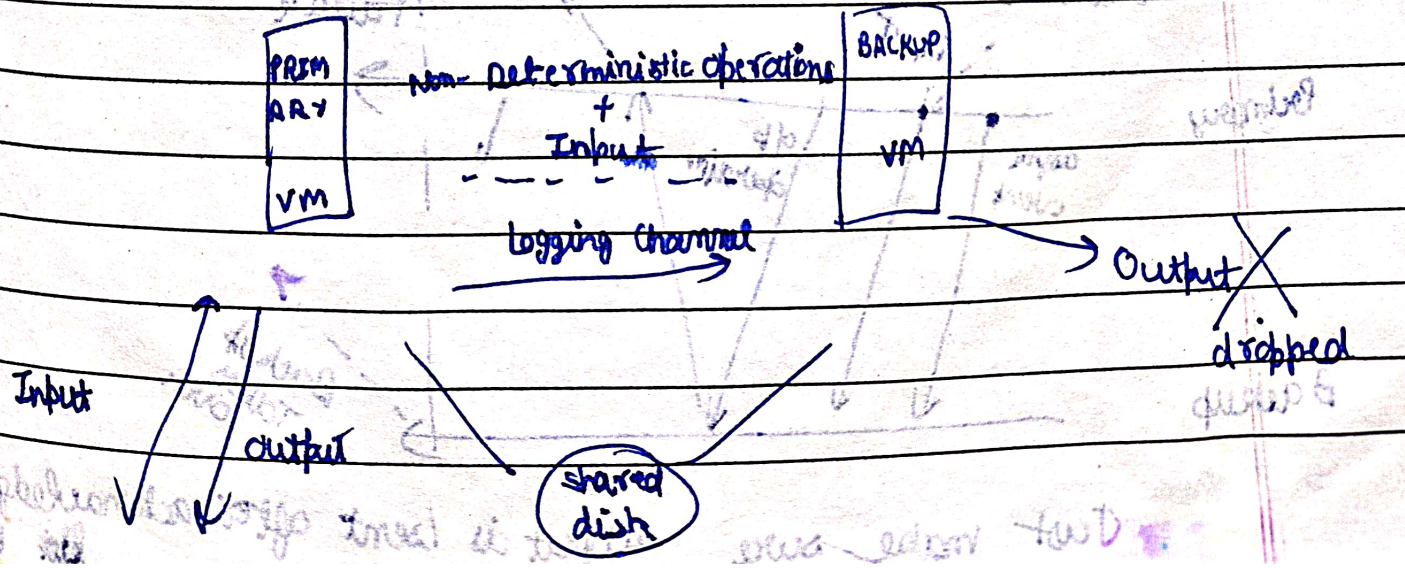
Bandwidth ↑

ii) Replicated State Machine: Just send same inputs requests to both the servers.

Non-deterministic operations such as clock, system interrupts can create trouble.

↑

A VM running on top of hypervisor can implement this approach.



1. Deterministic Replay Implementation

There are broad set of inputs, including network packets, disk reads, and input from the keyboard & mouse.
e.g. Virtual Interrupts, Reading clock cycle. (Non-deterministic)

Challenges
- Correctly capture all the input & non-determinism
- Correctly applying the inputs & non-determinism
- Doing so in a manner without degrading performance.

→ All inputs to primary VM are recorded & all possible non-determinism associated with the VM execution in a stream of log entries written to a log file.

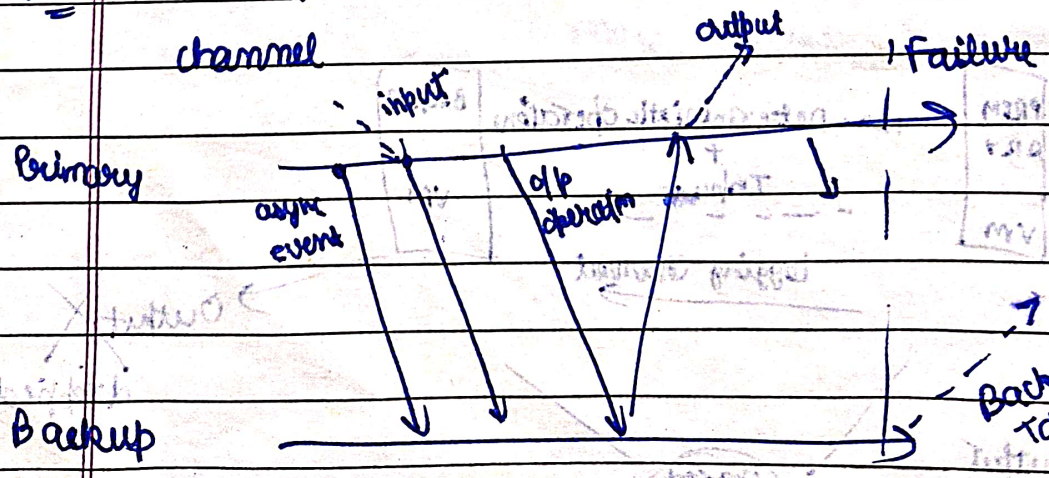
→ For non-deterministic operations, sufficient information is logged as well.

For events & interrupts, efficient event recording & event delivery mechanism is implemented.

2. FT Protocol

To ensure no data is lost when primary fails.

Note: The logs are not written to disk, instead sent via logging channel



Just make sure output is sent after acknowledging from backup

Forever
Page No.
Date : / /201

3. Detecting & Responding to Failure

Heartbeat + Monitoring (Log, Traffic)

To avoid split brain, it uses a atomic set & lock operation on shared storage. Thus, avoids it.

If backup dies,
primary will leave recording mode

If primary dies,
backup | waits until log entry is consumed
get a lock on shared storage.
gets another backup VM up

informs clustering service & set logging channel

Clustering Service

clones primary VM

Set up backup

STANF

Raft : An understandable Consensus Algorithm (2014)

Raft → Consensus Algorithm (works only on majority)

⇒ Abstract

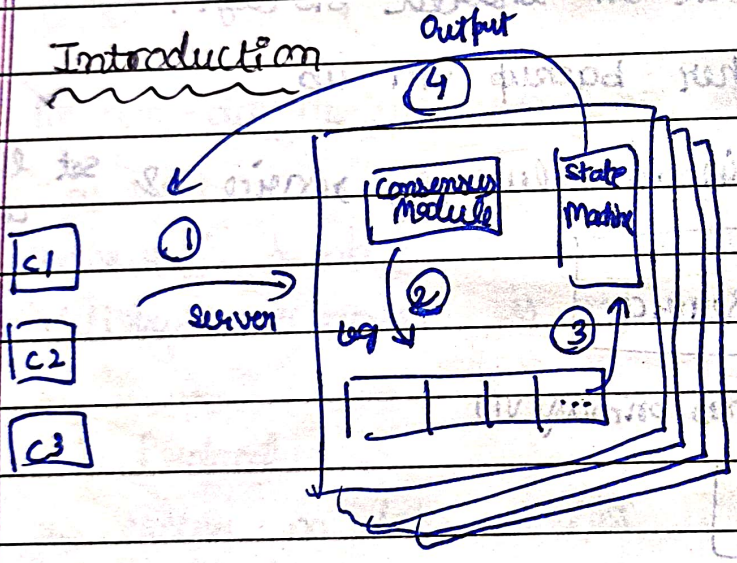
An easier version of Paxos

It seperates the key elements of the consensus, such as

- leader election
- log Replication
- safety

Understandability + Decomposition + State Reduction

⇒ Introduction



The consensus algorithm's job is to keep the replicated log consistent.

They (consensus algo.s) have the following properties

- They ensure safety (never incorrect results) under all non-Byzantine conditions, including network delays, partitions, and packet loss.
- They are fully functional as long as majority of servers are operational.
- They don't depend on timing to ensure consistency of the logs.

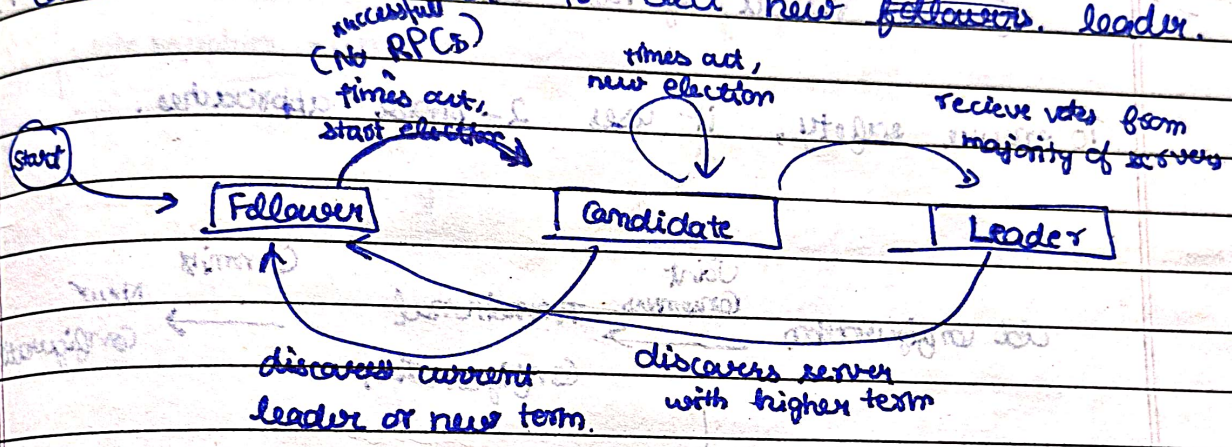
Forever
Page No.
Date : / /201

Algorithm

Basics

A lot of servers (typically 5)
Each Servers — Leader
— Follower
— Candidate

In normal state, 1 is leader and rest are followers.
- Leader does lot of things
- Followers respond to requests from leaders & candidates
- candidates are used to elect new ~~followers~~ leader.



Server States

Raft servers communicate using RPCs
- Request vote RPCs
- Append entries RPCs

* Leader only commits an entry once it has been replicated to majority of servers.

* During election, those will not get elected, who have lower commit index.

Note: Every term has atmost one term

cret LOL NERD.

Another requirement for RAFT is timing.

Broadcast time << election Timeout << MTBF

(All pings)

average time between failures for a single server

Cluster Membership Changes

Can be risked of two leaders for the same term.

To ensure safety, it uses 2-phase approaches.

old configuration → (Joint Consensus) → Transitional Configuration → (Commit) → New configuration

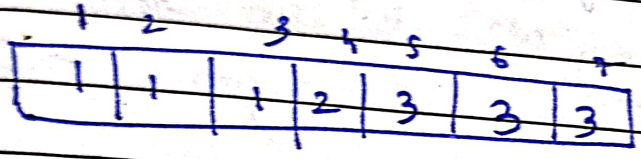
- i) Log entries are replicated to all servers in both configurations.
- ii) Any server from either configuration
- iii) Agreement (for elections & entry commitment) requires separate majority from both the old & new configurations.

Forever
No.

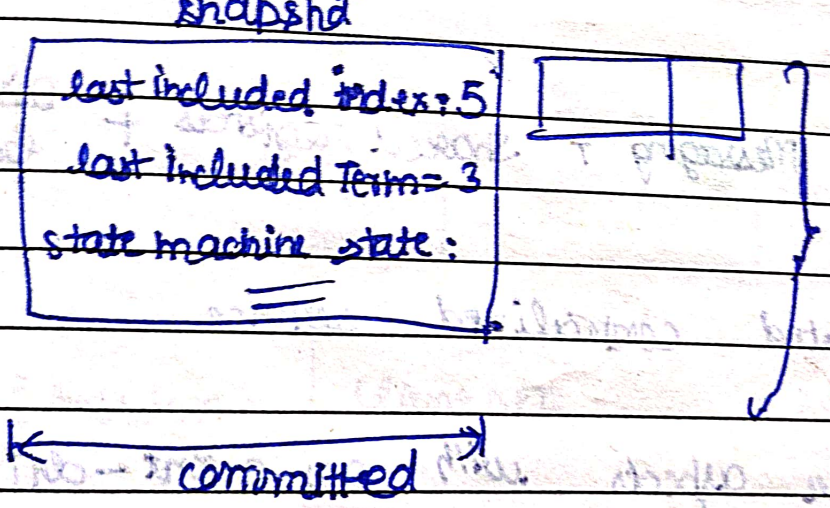
Forever
Page No.
Date : / /201

Log Compaction

Snapshot current system state & log up to that point is discarded.



log index before.



YAHOO

Zookeeper (2010)

Zookeeper → service to co-ordinating process of distributed systems/ applications

⇒ Abstract

↓

Elements of Group Messaging + Shared registers + ~~Distributed~~ lock service

in a replicated centralized service.

- It has wait-free aspects with an event-driven mechanism to provide powerful co-ordination service
- FIFO requests & linearizability for all requests.

⇒ Introduction

All distributed systems require co-ordination.

- Configuration
- Leader election/ Group Membership
- Locks

There can be several services for each of different co-ordination services

but...

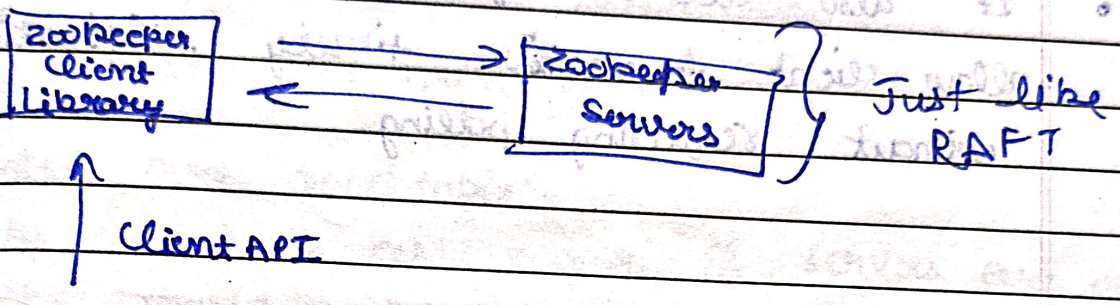
- Zookeeper implemented a co-ordination kernel and

Date : / /201

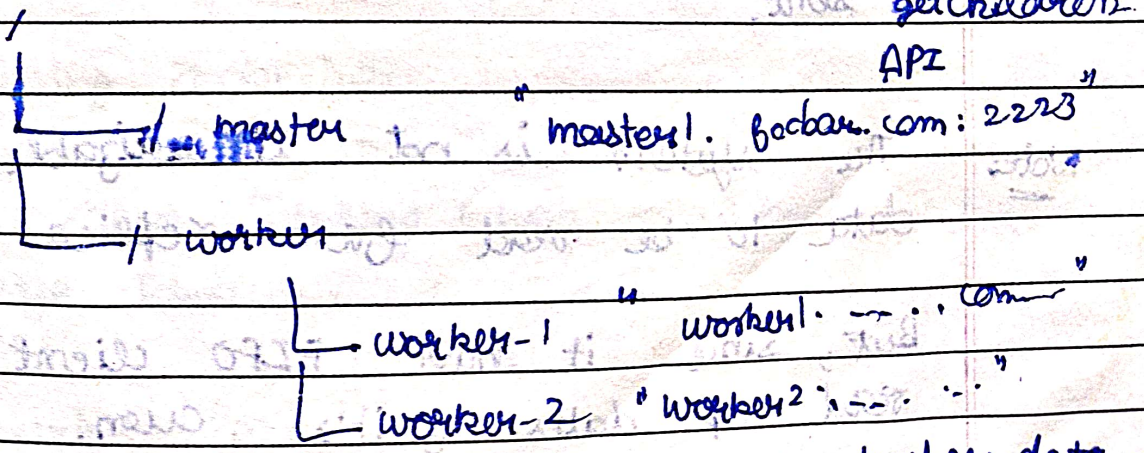
exposed an API that enables appl" developers to implement their own primitives.

It also avoided locks, and used wait-free data objects organized hierarchically as in file systems.

⇒ Zookeeper Service



- Zookeeper instead implementing a more general purpose system, it provides a file system.
- Replace concept of "file" with "znode".
and provide create, delete, exists, setData, getData, getChildren API



- znode is an in-memory data node of zookeeper data.

Date: / /

- There are two types of znodes, clients can create:
  - i) Regular
    - Persistent, creation & deletion explicit.
  - ii) Ephemeral
    - let system remove automatically

- It also allows sequential flag & watches to allow client to receive timely notifications of changes without requiring polling.

⇒ Zookeeper guarantees

i) Linearizable writes

All ~~Any~~ request that update the state of Zookeeper are serializable.

ii) FIFO client order

All requests from a given client are executed the same way that were sent.

Note: The system is not linearizable since it allows stale data to be read from replicas.

But, since it have FIFO client order, client will read up dates of it's own.

It also allows service to scale linearly

If $N_R \rightarrow N \times$ performance → scale better :)

↑
only for read-heavy appl^n though.

- This can be explained via

Leader config. update + Notification

"sync" operation (write before read)

So, guarantees are acceptable.

This works as long as majority of servers are alive (just like Raft)

⇒ Building Primitives

Zookeeper looks cool IMO.

It can do

i) Group Membership
ii) Efficient Lock Mechanisms
iii) Read/Write Locks

etc.

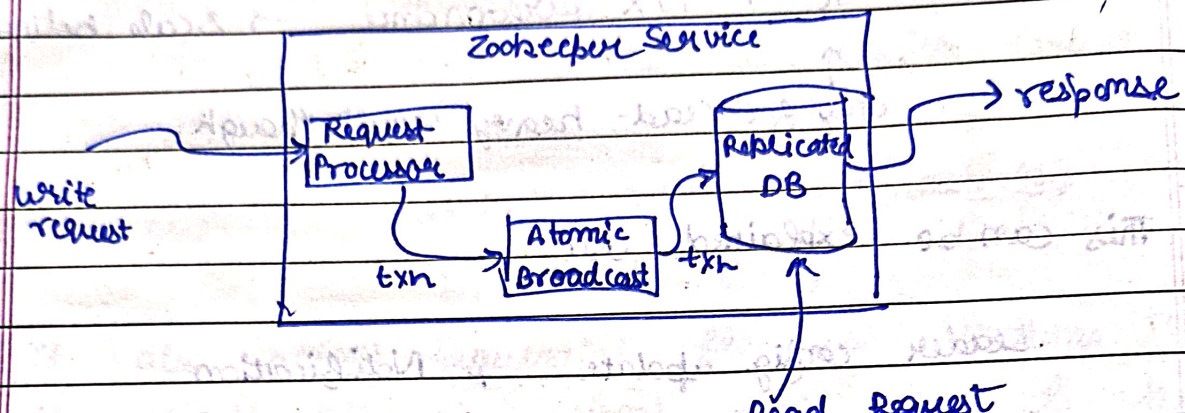
⇒ Applications using Zookeeper

i) ~~Zoo~~ Yahoo Fetching (web crawling) Service
ii) Message Brokers
iii) Kafka

& much more...

Forever
Page No.
Date : / /201

⇒ Zookeeper Implementation



Note: Zookeeper can deliver same messages twice, but I think they are idempotent.

i) Atomic Broadcast.

Update Zookeeper State → Leader → Broadcast

ii) Request Processor

SetDataTxn if success
error Txn if fail

iii) Replicated DB

In-memory
Uses snapshots to recover faster.

iv) Client-Server Interactions

On write request → Notifications → watchers

Read request → tagged with Zxid → FIFO order
(may return stale value)

UNKNOWN

Forever

Page No.

Date : / /201

Bitcoin (2009)

↓

P2P version of e-cash allowing online payments without having to go through a financial institution.

It have:

→ Network time-stamp transactions

→ Hashed into blocks

→ based on proof-of-work

⇒ Introduction

Motive

↓

→ Having computationally impractical mechanisms to reverse or commit fraud.

→ and allow routine escrow mechanisms.

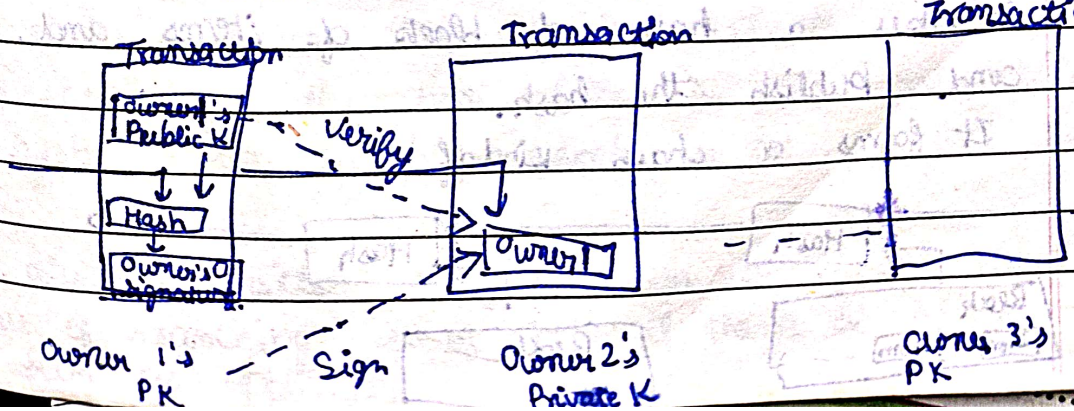
↓

both buyer & seller are satisfied

This paper solves double-spending problem.

⇒ Transactions

Electronic coin = Chain of digital signatures.



It is basically a digital signature stored on HDD.

→ Payer signs the hash of (public key + Previous transaction ↓ next of owner)

→ Payee can verify the transaction but don't know if payer double-spend the coin.

Approach 1: Use a Bank like system

✗ Not decentralized.

Approach 2: Use bitcoin system.

↓

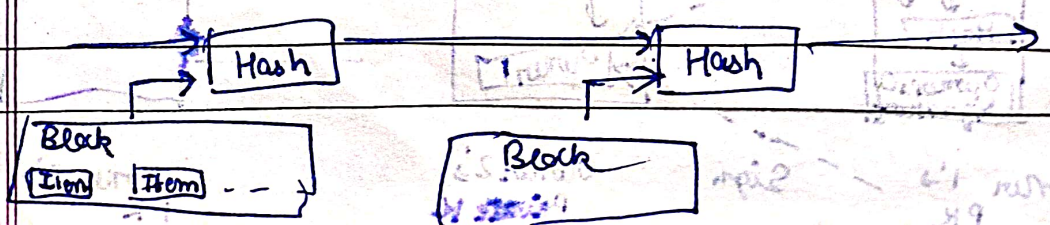
a system for participants to agree on a single history of the order which they were recieved.

⇒ Timestamp server

To avoid double spending, it introduces timestamp server:

It takes a hash of block of items and timestamp it and publish the hash.

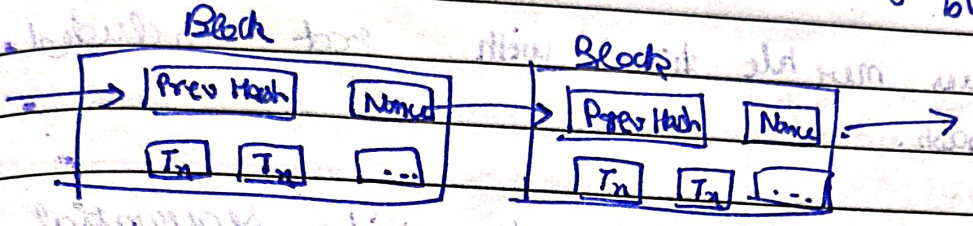
It forms a chain kindoff:



⇒ Proof of work

For a distributed timestamp server on a p2p basis, bitcoin uses proof-of-work.

add nonce to prev hash → starts with number of zero bits.



It avoids IP-based-one-vote system and uses computation-power-based government.

⇒ Network

1) New transactions are broadcasted to all nodes.
2) Each node collects new transactions into a block.
3) Each node works on finding a difficult proof-of-work for its block.
4) When a node finds proof-of-work, it broadcasts the block to all nodes.
5) Nodes accept the block only if all transactions in it are valid & not already spent.
6) Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

Nodes always consider longest chain to be the correct one & in case of branching saves both of them & select longest one as soon as applicable.

⇒ Incentive

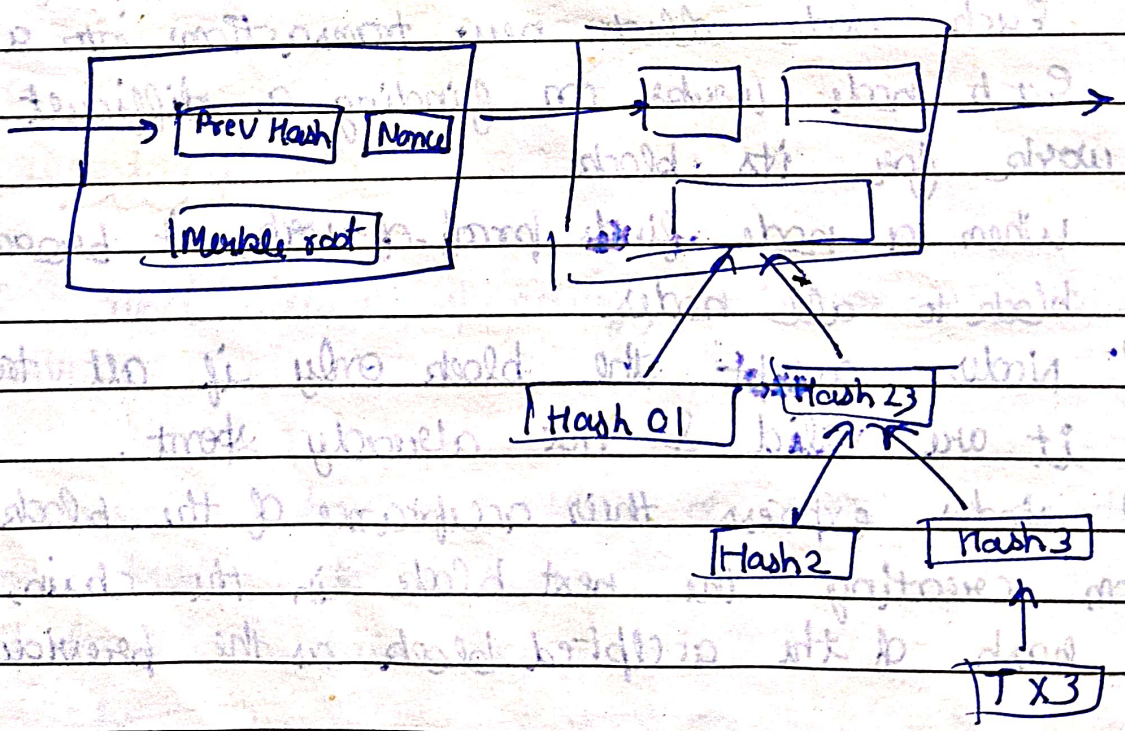
Halfs every 4 years.

+

Transaction fees

⇒ Reclaiming disk space

- Uses merkle tree with root included in block's hash.
- Allows less space and avoid sequential storing of transactions.
- A user only needs to keep a copy of the block headers of the longest proof-of-work chain, by querying network nodes.



Princeton University

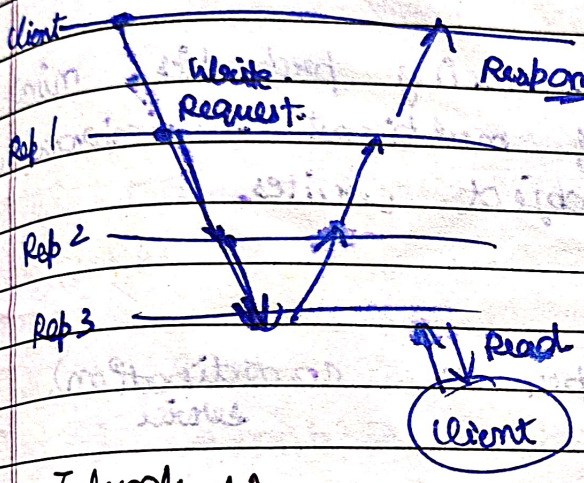
Object Storage on CRAQ (2009)

↓

Chain replication with Apportioned Queries.

⇒ Overview

A distributed object storage system focussed on improving chain replication & read throughput while maintaining strong consistency.



- It creates hotspot as reads goes through one node
- It have stronger consistency & cheaper write operations

⇒ Introduction

Object stores supports

↳ Read (Return data block under an object name)

↳ Write (change the state of a single object)

- These are like key-value DBs and are highly scalable.
- For this, object namespace is partitioned over many machines & each data object is replicated several times.

- Chain replication looks promising but only read throughput pulls it down, that's where CRAQ comes in.

YAY? Basically, it divides read operations over all nodes in a chain:

1. Supporting read operation while preserving strong consistency. (load balancing).

2. Allows eventual consistency among read operations and specify maximum staleness value.

3. Geographical load balancing properties, mini-transactions & use of multicast to improve write performance for large-object writes.

⇒ CRAQ (Uses zookeeper as coordination service).

It works as follows:

1. Each node in the chain can store multiple versions of an object: one clean version and a dirty version per recent write. All versions are initially clean.

2. When node recieves a new version of an object, it appends this latest version to its list for the object.
   - If not tail, mark as dirty
   - Otherwise, mark as clean (committed)

When ack message for an object version arrives at a node, the node marks object version as clean, and delete all prior versions of the object.

On read request

- If version (latest) is clean, return value.
- otherwise, contact tail & ask for tail's last committed version number.

This model allows CRAQ to support three forms of consistency

i) Strong Consistency (As above)
ii) Eventual Consistency (Don't ask tail for latest committed version no.)
iii) C.C with Maximum-Bounded Inconsistency (As the name suggests)

Problems with CRAQ:

i) If one node slow, affects all of the nodes in a chain
ii) We can also create multiple chains instead of CRAQ to have even more simplicity.

Amazon

Forever

Page No.

Date : / /201

Aurora (2017)

↓

Design considerations for High throughput cloud-native relational DBs.

⇒ Abstract

A relational DB service for OLTP offered as part of AWS (Transaction Proc)

- Since, today storage & compute is better today than network, aurora addresses this constraint.
- It also uses an efficient asynchronous scheme to achieve consensus without chatty & expensive recovery protocols.

⇒ Introduction

Well problem nowadays is network between DB tier and storage tier.

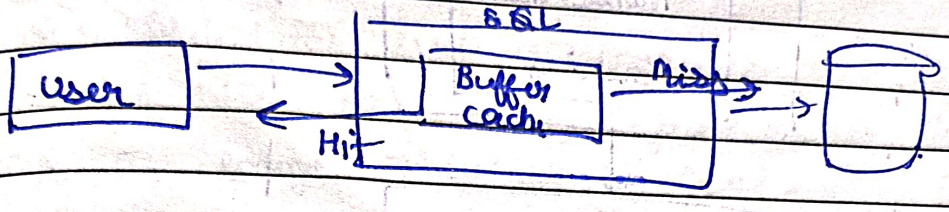
→ Before starting it, let's talk about internals of DB: (SQL)

- SQL server writes modified data page to the disk asynchronously:
  - i) Lazy writing → Inspect & remove infrequently used pages.
  - ii) Eager writing → during non-logged operations
  - iii) checkpoint → Periodic check to scan the buffer cache & write dirty page into the disk.
- On read request/ SQL query. (btw SQL server store data in 8 KB pages).

i) It locates the data & retrieves the data pages that contain those information.

ii) by retreiving it first from the disk to buffer cache

iii) Finally, returns result to the reader.



Read

- On write request, update query to be precise:

i) Locate page in buffer cache, if not found, retrieve into buffer cache from disk.

ii) Acquire locks on the page to be updated, update them now.

iii) Now, pages are called dirty page.

iv) This modification is recorded in transaction log and an ack is sent to the user. (changed page is still in buffer cache).

v) later, by one of the processes, changes are flushed into disk.

Write-ahead logging

→ Coming back now, this page miss or background tasks can cause context switches & resource contention

→ Amazon Aurora, addresses these issues by leveraging the redo-log across a highly distributed cloud environm

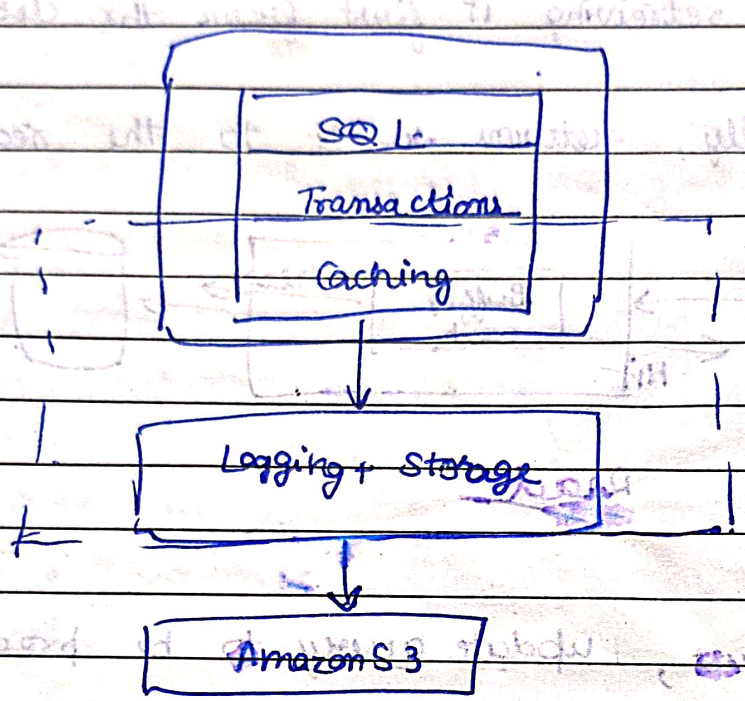
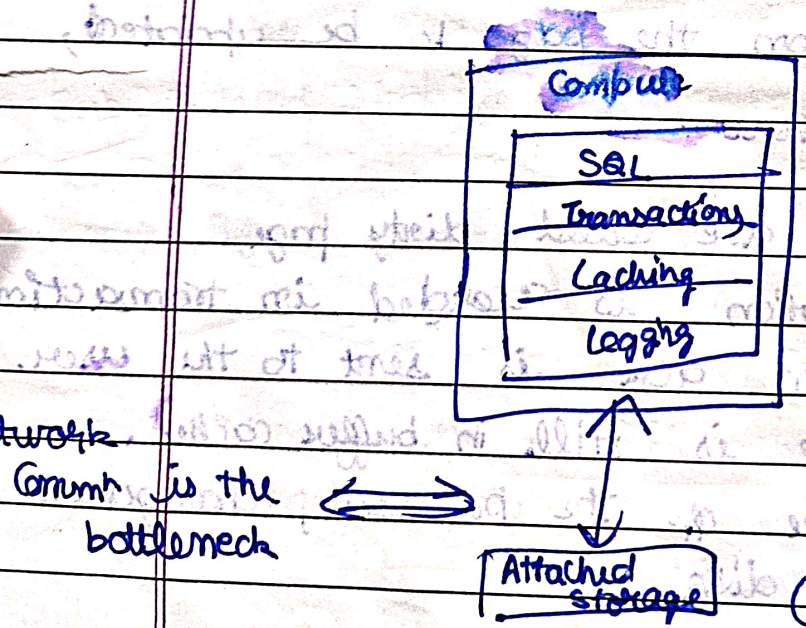


Fig: Move logging & storage of the database engine.



(Modern) Traditional DB Architecture

Network Comm^n is the bottleneck

(Fault tolerant)

→ The aurora architecture offers 3 major advantages

Forever
Page No.
Date : / /201

1. Protection of the DB from performance variance & transient or permanent failures.

2. Reduce network IOPS by only writing redo log records to storage.

3. Expensive operations such as (backup & redo recovery) are offloaded to continuous asynchronous operations across a large distributed fleet.

Sub 2: Durability.

Option 1: Replication

3 ways with 1 copy per AZ

use read/write quorums of 2/3.

Can it survive?

i) AZ Failure ✓ (2/3 still live)

ii) AZ+1 Failure ✗ (lose 2/3)

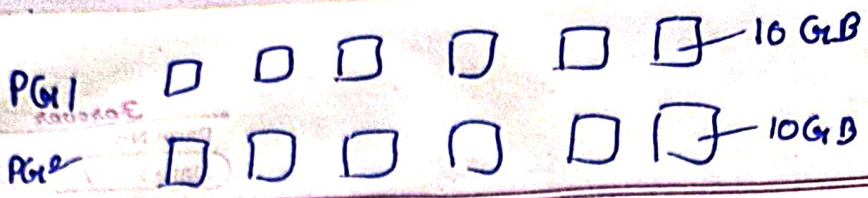
Option 2: Replicate 6 ways with 2 copy per AZ

write quorum of 4/6, read quorum 3/6

Can it survive

i) AZ failure (4/6 still live)

ii) AZ+1 failure (still have 3 copies)



Forever
Page No.
Date : / /201

Now, with the 6 copies, have to get the largest DB. size

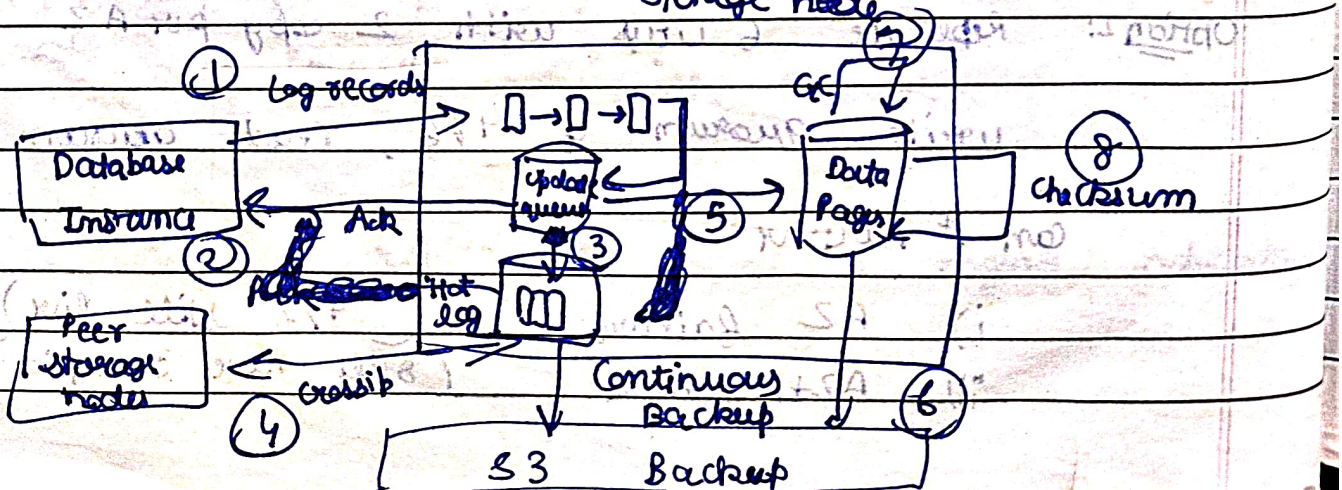
i) Partition volume into n fixed-size segments and replicate each segment 6 ways into a protection group (PG)

ii) If segments are too small, failures are more likely otherwise, repairs take too long.

iii) so, they choose 10 GB as segment size. since on a 10Gbps network line, it takes only 10 seconds to repair. (so, a total of 1 minute)

Sub 2? offloading the lower quarter of traditional DB to this tier. (LOG IS THE DB)

In traditional DB, replication leads to write amplification, so aurora, with its different architecture, offloads work by:



Forever
Page No.
Date : / /201

⇒ Normal operations

for quorum to work, it uses LSN (latest) to decide.

Log Sequential Number

I. Writes

VDL = Volume durable LSN (Trusted data for each replica)

VCL = Volume complete LSN (Guarantee Availability of all prior records)

LSN ≤ VDL + LSN Allocation Limit (Throttling purposes)

i) Database determine the PGs

ii) Sends LSN to all replicas in PGs

iii) On quorum of 4 acks, send respons to client.

II. Reads (No need for quorum though)

i) Same like traditional DB

ii) Establish a read-point & determine segment to get data from.

Or

Issue read-quorum & get the latest version data.

Systems Research Center

Forever

Page No.

Date : / /201

Frangipani (2000)

↓

A scalable distributed FS

⇒ **Abstract**

An ideal Distributed FS would provide

i) Shared access to the same set of files

ii) Scalable

iii) Availability is high

iv) Minimum Human administration.

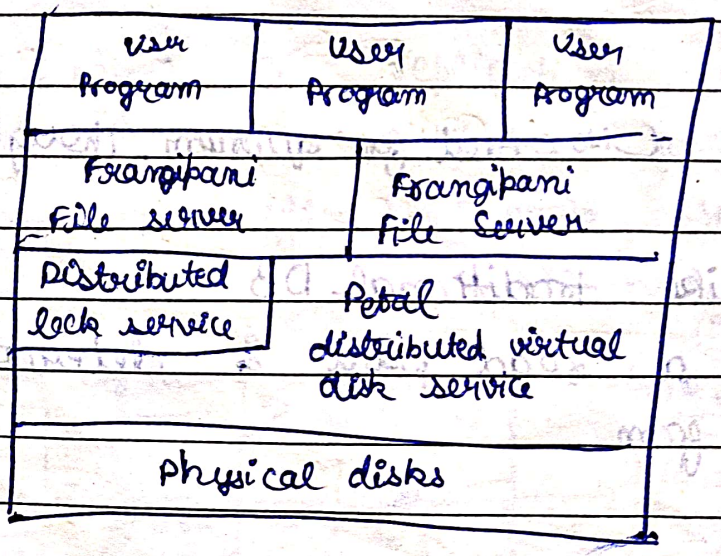
It has a fairly simple set of features.

Frangipani approximates this ideal, by a two layer architecture :

Architecture

- Lock Service — Upper layer ↓ Machines.
- Petal — lower layer ↓ Distributed storage service

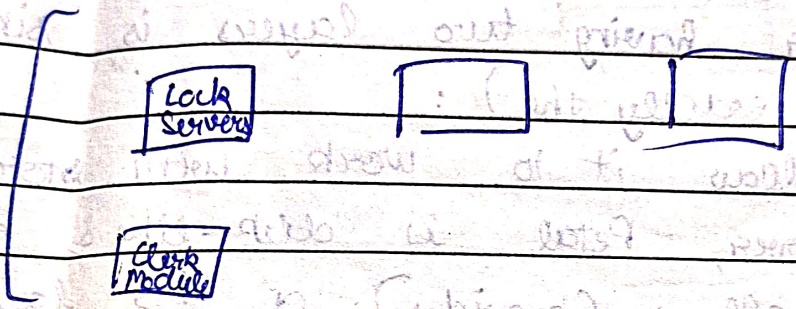
⇒ **Introduction**



⇒ Lock Service

- Multiple reader / single-writer
- All locks are sticky, i.e., a client will retain a lock until some other client needs a conflicting one
- Based on lease, 30s on start, renew later on.

Lock Service



Frangipani

- Uses fault-tolerant, distributed failure detection mechanism to detect the crash of lock servers.
- i) When a frangipani server crashes, the locks that it owns can't be released until recovery have been performed.
- ii) So, lock service asks clerk on another frangipani machine to perform recovery & then release all locks belonging to crashed server.

→ FGP file servers can be scaled up & uses locks for co-ordinating their actions.

→ Petal & the lock service are also distributed for scalability, fault tolerance. They provide large virtual disks (replicated).

→ FGP file server module uses Petal device driver for every comm^n required.

- The reason for having two layers is similar to that of Aurora (not exactly the) :
  - → To allow it to work with storage abstractions.
  - → Moreover, Petal is disk-like & not file-like, FGP allows (provides) a FS layer.

Disk Structure

FGP uses the large, sparse disk address space of Petal.

→ A Petal virtual disk has 2^{64} bytes of address space.

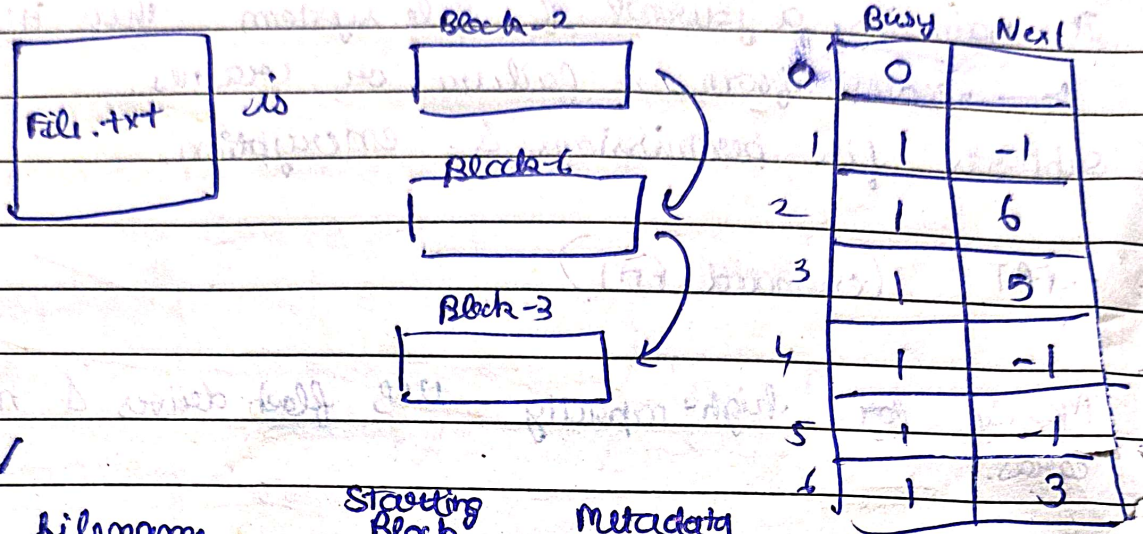
- If file is written, petal commits physical disk space to virtual address.
- Petal also provides a "decommit" primitive, that frees the physical space.

Each commit size is 64 KB of chunk

Date : / /201

→ Before starting it, let's talk about FSs:

i) FAT (File Allocation Table)



filename	Starting Block	Metadata
file.txt	2	—

(Directory Table format)

Note: Root file directory has a fixed address, so we know how to get started.

FAT 12, 16, 32 are (represents) the bits in each table element of FAT.

So,	Max file size	Max Volume size.
FAT12		
FAT16		
FAT32		

ii) NTFS (New technology file system)

→ File size limit = 16 EB (exabytes)

→ It maintains a journal of file system, thus it's able to recover from a failure or crashes.

→ supports file permissions & encryption.

iii) exFAT (extended FAT)

→ Mainly for high-capacity USB flash drives & memory cards.

iv) ext4

→ Linux dfault FS

→ No native windows or MacOS support.

on the other hand, ~~windows~~ (linux) ubuntu can read-write NTFS

v) HFS, HFS+ & APFS ← Apple file system

↓

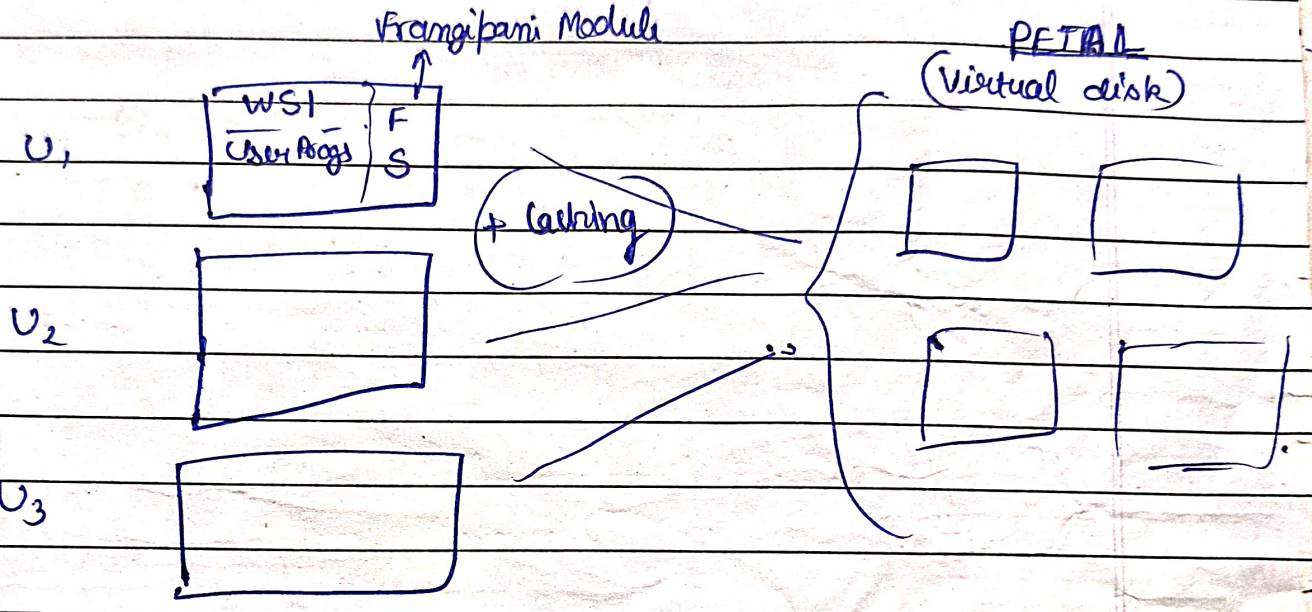
→ Hierarchical file system for MacOS

→ No native windows or ~~linux support~~.

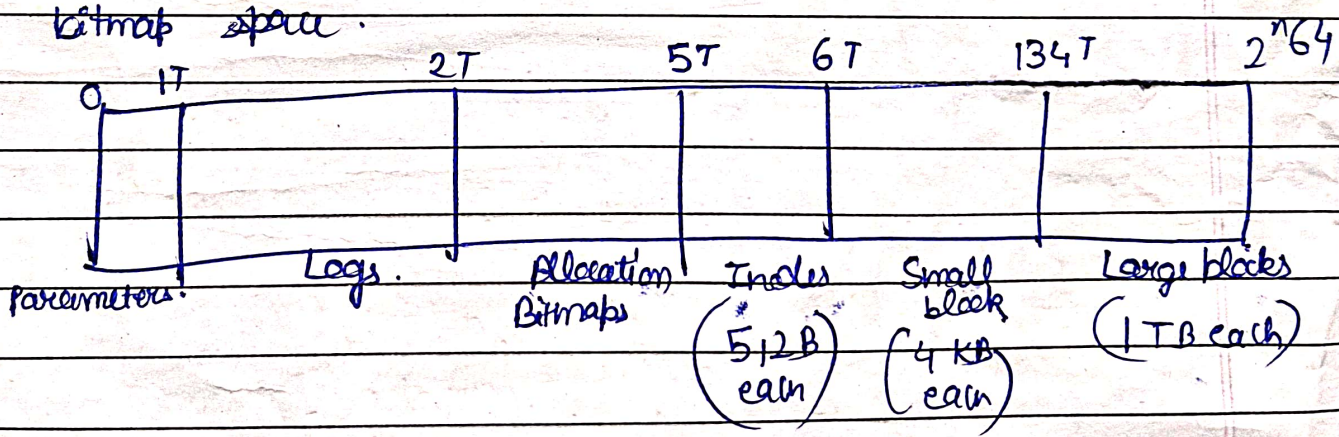
Forever
Page No.
Date : / /201

Summarizing

i) Cache- coherence
ii) Public client Logs
iii) Frangipani version numbers



Each server has its own log & its own blocks of allocation bitmap space.



So, to have caching & a decentralized platform, we need to work upon

a) Cache coherence
b) Atomicity
c) Crash Recovery.