Distributed Systems

> Author: Guanzhou (Jose) Hu 胡冠洲 @ UW-Madison CS739
>
> Teacher: Prof. Michael Swift

Introduction

A **distributed system** is a collection of *independent*, *autonomous* hosts connected through a communication *network*, collaboratively providing a uniform *service* to users.

Properties

A distributed system is desired to have the following four properties (these are reasons why people are interested in using a distributed system; otherwise, a monolithic server is always a better choice):

1. *Fault-tolerant*: allows component failures without revealing any incorrectness
2. *Highly-available* (*Reliable*): can resume providing services even when some components have failed
3. *Scalable*: can scale "out" (not "up") to a larger volume/serve more users without significant performance degradation
4. *Physical distribution*: distributed geographically, letting local users communicate faster with each other

However, the distributed nature will make many things harder to provide compared to a monolithic setting. To build a good distributed system, these are the additional internal properties that system designers must keep in mind:

- *Recoverable*: failed components can restart & rejoin the system

- *Consistent*: in presence of concurrency & failure, can coordinate components and provide a "right" answer, based on some definition of what is considered "right"
- *Predictable performance*: provides desired responsiveness in a timely manner
- *Secure*: authenticates user access to data and services

Kinds of Failures

Being tolerant to **failures** is one of the main reasons people wanna use distributed systems. These are some categories of failures we are interested in:

- *Halting* failures: a component stops. No way to guarantee detection/notification of whether/when it stopped
  - *Fail-stops*: special, clean kind of halting failures where it is assumed that a component will always send notifications when it stops
- *Network* failures: a network link gets congested or breaks completely
  - *Omission* failures: a message gets discarded due to congestion/bad link, without notification to either side
  - *Partitioning* failures: special kind of network failures where the topology breaks down to two or more disjoint sub-network partitions
- *Timing* failures: some temporal property of the system is violated, e.g., timeouts, or deviating clocks
- *Byzantine* failures: the hardest scenario - components within the system may corrupt data/modify messages/drop data on purpose, including being attacked by malicious programs. The only guarantee is cryptographic math still hold

Sharding, Replication, & Scalability

Grapevine

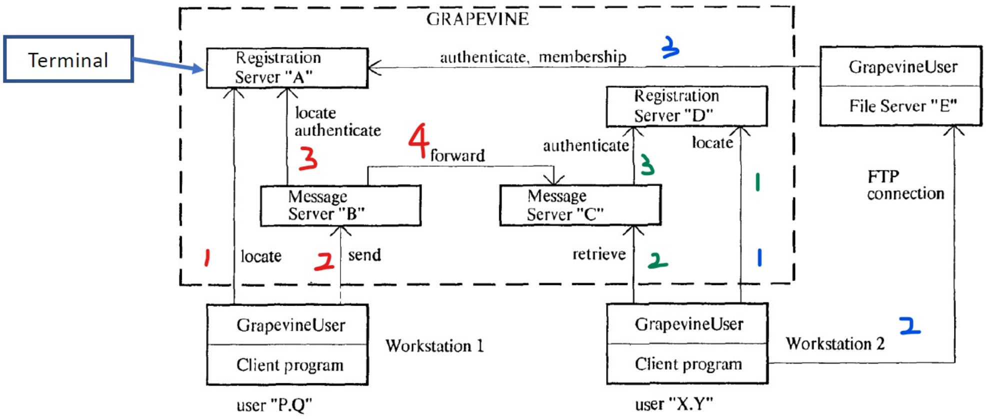
Link: https://users.soe.ucsc.edu/~sbrandt/221/Papers/Dist/schroeder-tocs84.pdf

Motivation

- To *scale out* by adding more machines instead of *scale up* by making more powerful machines
- Users distribute geographically around the globe, want local machines in different cities
- Availability: keep service running when machines go down
- Targeting an environment of mixed LAN & WAN networks with low reliability

Contribution

- A messaging system where users send emails to users



  - Each machine runs *a message server* + *a registration server*
  - Red - user sending a message
  - Green - user checking its inbox
  - Blue - user accessing a remote service which uses Grapevine as a location/authentication mechanism
- Highly-available: guarantees that if any machine is running, the service is up
  - Replication:
    - Each registry is replicated on multiple machines
    - Primary & secondary inbox for each user; Tertiary inbox at the other end of unreliable link
    - Functions of all services run on every machine - machines are *homogeneous*
  - Relaxed consistency:
    - Allow users to see a partial/stale inbox

        - Messages are asynchronously delivered - do not need the entire system to do the messaging
    - *Idempotent* operations:
        - Messages/Registration operations are "ID"ed and can be performed multiple times without hazards
- Can scale out to a large number of users across the globe:
    - Partitioning (Sharding):
        - Registration info split up into small registries
        - Registry sizes are kept small - scale by making more registries
        - Making an indirect hierarchy for large distribution lists; Move distribution list expansion to multiple servers
    - Caching:
        - Buffer authentication results to boost performance
    - Locality:
        - Group frequently communicating users into registry (manually)
        - Allow local users to communicate over local links
    - Don't replicate large objects; Delta replication
- Attempts to do load balancing:
    - Put secondary inboxes of different users onto different machines
    - Manual assignment of registries to machines, according to domain knowledge
    - *Admission control*: rejects requests when disks almost full, reserves idle capacity for fail over
- Allows decentralized administration: expert could manage the system remotely

Drawbacks

- *Eventual consistency*: if we wait infinitely, up-to-date messages should eventually go through
    - But users can see partial/stale data
    - Message content is not replicated - only the registration metadata
    - No consensus algorithm: needs manual recovery from replicas

> "Stupider the program, the stronger consistency we may want". Human are more tolerable to inconsistent results compared to strict programs.

Giant-Scale Services

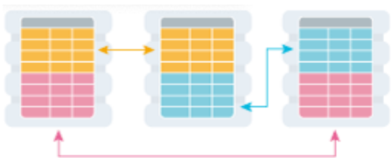
Link: https://courses.cs.washington.edu/courses/cse454/05sp/papers/GiantScale-IEEE.pdf

Motivation

- Models the design of datacenters serving a huge number of users from the Internet
- Workload properties of giant-scale services:
    - Read mostly, e.g., web search, forums, social media, ...
    - Interactive (query-driven)
    - Short requests, won't crash in the middle
    - Can give back flexible (partial) results, e.g., web search, mails, shopping list, ...

Contribution

- System architecture modeling of a giant-scale service site (Fig 1.):
    - Load balancing techniques:
        - Round-robin DNS: distributes different IP addresses for a single domain in round-robin
            - Balances load well, but does not hide inactive servers
            - Ignores cache locality on backends
        - Switches:
            - Layer-4 switches: understand TCP & port numbers
            - Layer-7 switches: understand application-level (HTTP) requests; can detect down nodes in this case
    - Assumes backends are connected through a *backplane (backbone)* network and do both replication + partitioning



- Proposed several availability metrics:
  - Uptime = (MTBF - MTTR) / MTBF

    To improve uptime, either increase MTBF or reduce MTTR
    - Increase MTBF: more reliable hardware/software
    - Reduce MTTR: faster reboot; faster reconfiguration; faster initialization; be stateless
  - Yield = #queries completed / #queries offered
  - Harvest = size of data returned / total amount of data should've returned
- DQ principle:
  - D := amount of data accessed per query; Q := number of queries served per time unit
  - DQ = D $\times$ Q = bandwidth of cluster available, largely fixed in hardware

    One machine's failure reduces DQ by a machine's worth
  - Replication: failure reduces Q
  - Partitioning: failure reduces D
- Overload after failure: $\frac{n}{n-k}$
  - Replication: load from failed machine distributed to remaining ones - need to reserve capacity to handle failures
  - Partitioning: just reduce D
- The 3 ways of doing online upgrades have the same DQ loss (Fig 5.)
  - *Fast reboot*
  - *Rolling upgrade*
  - *Big flip*

LARD

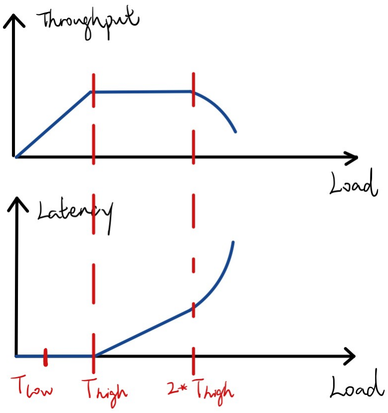
Link: https://www.cs.rice.edu/~alc/comp520/papers/p205-pai.pdf

Motivation

- Targets a web server cluster scenario where you have 1 frontend node dispatching connections to N backend nodes
- *Caching* is significant: popular content follow a *Zipf curve* $pop(x) \propto \frac{1}{x^\alpha}, \alpha > 1$
  - So we'd better send requests for an object to the same backend, to make cache on the backend hot
  - Previous load balancing policies do not consider locality
    - Weighted round-robin (WRR)
    - Hash-based dispatching; Consistent hashing
    - Multicast
    - Random mapping
- Can respond to changing workloads, dynamic

Contribution

- Combines locality-aware content distribution with load balancing (Fig 3.)
  - Distributes requests on each piece of target data to only a subset of backend servers to improve cache locality
  - Dynamically adjusting the assignment list for each target based on backend load to achieve load balancing
  - Load is measurement as #open TCP connections
- Load behavior of the system:



- Hot objects may need more than one cache, so should do LARD with replication
  - Add new backends periodically if least-loaded server n's load is
    - $> 2 \cdot T_{high}$, then no matter what, add a new one to help
    - $> T_{high}$, and if some node is $< T_{low}$, add it to help
  - Remove most-loaded replica if set hasn't changed for a while; make threshold time K long enough to prevent *thrashing*
- Designs a TCP handoff protocol to allow the frontend peek packet content then forward the connection to a backend

Epidemic Algorithms

Link: http://bitsavers.trailing-edge.com/pdf/xerox/parc/techReports/CSL-89-1_Epidemic_Algorithms_for_Replicated_Database_Maintenance.pdf

Motivation

- Xerox wanted a globally-replicated database of a huge number of small-sized machines
  - Each update is injected at a single site and must be propagated to other sites
  - Flooding may cause way too much traffic (300 sites $\rightarrow$ 90000 messages per night)
  - Also wants short time to propagate to all sites
- Fundamental problem: replicating data to many machines across wide area
  - Very large number of participants
  - Machines/links may fail and come back
  - Network topology not uniform
- Biggest idea: use *randomness*, rely on *probabilities* to propagate

Contribution

- Categorizes three types of communication techniques (Sec 0.):
  1. *Direct mail* (direct *flooding*): each update immediately mailed to all other sites - this is NOT epidemic
     - Cons:
       - Node may not know everybody else
       - Mailing message can fail
       - Overwhelming amount of traffic
  2. *Anti-entropy*: site regularly chooses another site at random, and the two exchange database contents to resolve any differences between the two
     - Pros: complete sync of all info
     - Cons: expensive to run, may need better data structures to reduce data transmission
  3. *Rumor mongering*: when a site receives a new update it becomes a "hot rumor"; while a site holds a hot rumor, it periodically chooses another site at random and ensures it has seen the update; when a site has tried to share a hot rumor with too many sites that have already seen the update, the rumor becomes cold without further propagation
     - Pros: less traffic
     - Cons: some sites could miss the information (i.e., has residue); must back it up with above two mechanisms
- *Push* vs. *Pull*, active DB vs. quiescent DB (Page 10.)
  - Let p_i be the probability of a node remain susceptible after round i
  - Pull follows $p_{i+1} = p_i^2$, works better for active DB where p_0 is small; good for ending a rumor
  - Push follows $p_{i+1} = p_i(1 - \frac{1}{n})^{n(1-p_i)} \approx p_i e^{-1}$, works better for quiescent DB; good for starting a rumor quickly
- Three metrics for rumor mongering (Page 9.; Tab 1.,2.,3.):
  - *Residue*: how many nodes untouched after the end of the rumor
  - *Traffic*
  - *Delay*: average vs. last

Drawbacks

- Can at best achieve eventual consistency

Chubby

Link: https://static.googleusercontent.com/media/research.google.com/en//archive/chubby-osdi06.pdf

Motivation

- Needs a service that is easier for applications to use - *locking* is natural
- Must support a huge number of concurrent clients

Contributions

- A distributed locking service running with small, odd number of servers over a consensus protocol (Fig 1.)
  - Organizes locks (which are small files called *nodes*) as a UNIX file system semantic
    - But does not maintain info like last modify time
  - Provides distributed locking: servers agree on which lock node is now acquired by which client
    - Reader lock can be held by many clients
    - Write lock can be held by at most one client at a time
    - Locks are *advisory* - not holding a lock does not prevent you from accessing the resource it protects; so assumes client libraries are honest, but allows more flexible administration
  - Locks are just small files, so clients can communicate through some small data in the lock file
- Client cache session states locally for much better performance and scalability:
  - Sends periodic `KeepAlive` RPCs to the Chubby master
  - Master replies with *notifications* - refreshes lock *leases*
  - *Consistent caching* achieved by master including *cache invalidations* in the notifications; only when a client acks this invalidation can a master grant the write lock to another client

    Turns out to be a great fit for a name service, and a better approach than the current TTL-based DNS caching
- Lock *sequencer number* progress every time the lock is given to a different client - avoids a timed-out client waking up later and tries to use the lock it used to hold
- On master fail-over, client times-out a `KeepAlive`
  - Client now cannot use the locks
  - Client starts a *grace period*, keep sending `KeepAlive` RPCs to Chubby
  - If a new master is now elected, new master acks a `KeepAlive` and grants a new lease to the client

Stronger Consistency & Consensus

Chain Replication

Link: https://www.cs.cornell.edu/home/rvr/papers/OSDI04.pdf

Motivation

- Need replication for fault tolerance, not for performance
- Assume a *replicated state machine* (SMR) model, and tries to provide strong consistency (global ordering)
  - Replicas need to agree on which put requests have been completed
  - Completed puts must take effect on all replicas and to all subsequent client gets

Contribution

- Use a *chain* topology of machines, is a variant of *primary-backup* (Fig 2.):
  - Puts arrive at the head node, propagate through the chain to the tail, completes when acknowledged by the tail
  - Gets all served by the tail node - so returns the latest value seen by the tail at this point (*linearizability*)
- When machine fail-stops, the monitoring service will detect it by timeout and removes it out of the chain (Fig 3.)
- New replica added to be the new tail
- To tolerate f failures, need $f+1$ servers

Drawbacks

- Chain replication itself is not complete when you need membership changes - you will need some external Raft/Paxos consensus to determine who is the head node & the chain topology; clients talk to this monitoring service to find out this info as well
- NO network partition tolerance

> A typical critique is that strong-consistency consensus algorithms seem to make one leader node the performance bottleneck. However, this is not a fair argument: even if we have some algorithm that does not have a central leader, to achieve consensus, every request still must be seen by all participants - so overall we cannot yield better than one machine's throughput.

Logical Clock

Link: https://lamport.azurewebsites.net/pubs/time-clocks.pdf

Motivation

- Physical time synchronization across distributed nodes is hard
  - *Drift* of time advancement hardware
  - *Skew*: constant difference between machines
  - The Internet is asynchronous & best-effort
- What we really care about is the *ordering* of the *events* themselves, not an accurate time value

Contribution

- Defines the *happens-before* relation ($\rightarrow$)
  - Each process is a sequential program, sending/receiving messages
    - If a and b are on the same process and a comes before b, then $a \rightarrow b$
    - If a and b are the sending and receiving of a message, then $a \rightarrow b$
    - The relation is transitive
  - If we cannot infer the order between two events, we say they are *concurrent*
- Introduces *logical timestamps* C: if $a \rightarrow b$, then $C(a) < C(b)$
  - Process increments its clock on every local event
  - On receiving a message, progress my local timestamp to be $\geq$ the message sender's timestamp if not so
  - This is a *partial ordering*; A *total ordering* can be achieved by breaking ties by e.g. PIDs

Distributed Snapshot

Link: https://dl.acm.org/doi/pdf/10.1145/214451.214456

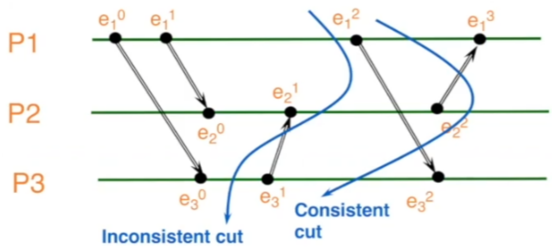
Motivation

- Detecting a *global property* (*snapshot* value) in a distributed system
  - No reference to an object so it can be GC'ed
  - Deadlock detection
  - Computation termination
- The property must be a consistent one agreed by all nodes, but can be a *predicate property* (*stable property*), i.e., only happens once per phase in the system
  - Once the property becomes true, it remains true
  - We don't need to worry about how to kick off the next phase

Contribution

- An algorithm to find a *consistent cut* of a distributed system

  A cut is inconsistent if there are events e_i, e_j s.t. $e_i \rightarrow e_j$ and $c_i \rightarrow c_j$, but $e_i \nrightarrow c_i$

- Defines (Fig 1.) global state := {process states} + {channel states}
  - Event is a local state transition on one process, may produce or consume a message
  - The system as a whole is a big state machine transiting between states (Fig 4.,7.); to reach a state S from initial state, there might be many possible paths of *computation*
- The snapshotting algorithm:

- Sending rule: p sends one *marker* along c after p records its state and before it sends further messages along c
- Receiving rule on q receiving a marker through c:
  - If q has not recorded its state, then q records its state, and then records c's state as empty
  - Else, q records the state of c as the sequence of message received along c after q's state was recorded and right before q received the marker

Drawbacks

- Did not talk a lot about how to collect the snapshotted global state from all processes
  - Write to a third-party shared FS?
  - Broadcast by flooding?
  - Put initer's name in the initial marker?
- Recorded state might be one that never happened in real world ordering, e.g. Fig 7. if p sends marker right after M
  - But it is OK because it is a consistent cut - we can safely use the snapshot and still infer the correct property

Two-Phase Commit (2PC)

See https://en.wikipedia.org/wiki/Two-phase_commit_protocol.

- Safe, but poor *liveness*: all participants wait indefinitely when coordinator fails in the second phase
  - Optimized *server termination protocol* (Slide 20.,21.)
- Must use a durable *write-ahead* log on storage to survive across crashes

Raft

Link: https://raft.github.io/raft.pdf

Motivation

- *Consensus* requirement on multiple nodes agreeing on the same agreement:
  - *Termination*: the procedure must eventually decide on one value
  - *Agreement*: all processes agree on the same value
  - *Validity*: the value that has been decided must have been proposed by some process
- Useful for cases where we do need strong consistency and some availability, but do not need very good performance

Contribution

- Strong consistency using the *majority rule*; To tolerate f failures, need $2f+1$ servers
- Assumes servers use a log and builds the consensus algorithm directly over a replicated log
- I will omit more about Raft. Please see the paper (Fig 2.) and here.
  - Possible follower states (Fig 7.)
  - Why cannot commit on entries from older terms but must wait for the commitment of something in up-to-date term (Fig 8.)
  - Membership change is sent as a log entry, needs majority of both in the middle period (Fig 10.,11.)

Paxos

Link: https://lamport.azurewebsites.net/pubs/paxos-simple.pdf

Motivation

- The ancestor of all consensus protocols

Contribution

- Strong consistency using the *majority rule*; To tolerate f failures, need $2f+1$ servers
- First proposes a *single-decree* Paxos algorithm for agreeing on a single value:
  - Have *proposers*, *acceptors*, and *learners*
  - Phase #1 - *prepare* phase: proposer selects proposal number n and tries to get acknowledgement from majority of acceptors; acceptor replies with the highest proposal number and value it has accepted
  - Phase #2 - *accept* phase: proposer sends accept request on the highest-numbered (say n') value among responses; acceptor accepts if n' is still up-to-date in all prepares it has seen, and notifies the learners
  - See the three cases of prepare in Slide 31.,32.,33.

- To reduce message complexity: use a *distinguished learner* to listen from acceptors and broadcast to other learners
- To reduce the occurrence of *livelocks*: elect a *distinguished proposer* that is the only one making proposals
  - Liveness issue example in Slide 34.
- A complete *multi*-Paxos algorithm can be built upon this if we need a log of commands like in Raft:
  - Each server acts as a proposer, an acceptor, and a learner
  - System elects a *leader* who acts as the distinguished proposer + distinguished learner; Runs multiple instances of the single-decree algorithm, one per client command

PBFT

Link: http://pmg.csail.mit.edu/papers/osdi99.pdf

Motivation

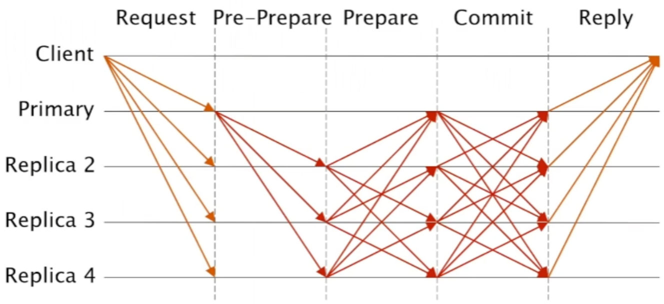
- *Byzantine faults* - the *Two-general paradox*: if you can have faulty node or message contents can be corrupted, you can never safely achieve an agreement with only two/three nodes

  Simplified recursive proof (Slide 7.)
- Previously we just assumed fail-stops. Now, if we can have Byzantine faults, we will need a more complicated protocol
  - Can be benign causes like hardware failure, bitflips
  - Or can be malicious causes like an attacker taking over several nodes

Contribution

- BFT; To tolerate f failures, need $3f+1$ servers

  Basic idea behind $3f+1$ is that any two $2f+1$ quorums will overlap in $f+1$ nodes, so you can tolerate f failures inside the overlap but still have the two groups achieve agreement
  - Each of the two $2f+1$ quorum agrees on a value using the majority rule
  - There must be at least one honest node that is in both quorums, so the two agreed values must be the same
- Use digitally-signed messages to ensure that honest nodes produce verifiable messages
- Picking the number of failures f to tolerate: Slide 27.
- I will omit more about PBFT. Please see the paper.



Drawbacks

- To achieve BFT, we need to exchange a huge traffic of messages - performance is terrible

> Interesting critique against BFT papers: https://www.usenix.org/system/files/login-logout_1305_mickens.pdf.

Blockchain

Link to Ethereum whitepaper: https://ethereum.org/en/whitepaper/

- Blockchain can be thought of as a stochastic, decentralized approach to Byzantine fault tolerance
- *Sybil attack*, why VMs can break the majority requirement, why we must need *proof of time/computation/work*
- I will omit more about Bitcoins. See the whitepaper and here.

Availability & Failure Recovery

Dynamo

Link: https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

Motivation

- Normal *relational* database not the right fit for some applications
  - Strict consistency too expensive
  - Many expensive & complicated features not needed: join, transactions, ...
  - Hard to scale
  - Hence, sometimes just need a *NoSQL* key-value database
- Not all things in the ACID principle are needed:
  - *Atomicity*: yes
  - *Consistency*: no - just need eventual consistency
  - *Isolation*; N/A (updates just in one-key granularity, no transactions)
  - *Durability*: yes
- Highly scalable, P2P, no master
- Tolerate almost any kinds of failures
- Meets 99.9% latency requirements

Contribution

- Defines *service-level agreements* (SLA):
  - Expected load
  - %-ile latency under load
  - % availability
- Dynamo resolves version conflicts during *read* instead of *write* - it provides this API interface:
  - `get(key)` $\rightarrow$ `(list, context)`, a list of objects with conflicting versions

    If there is conflict, client needs to resolve the conflict and tells Dynamo the result in its next `put`
  - `put(key, value, context)`
  - Client doesn't need to know all nodes, just any node
- Uses *consistency hashing* to handle data placement (Fig 2.)
  - `hash(X) mod k` is uniform and fast, but does not handle nodes leave/join elegantly (need to move a lot of data on config changes)
  - Consistent hashing hashes machines & objects all onto a circular namespace

    Uses *virtual nodes* to map one machine to multiple places on the ring for better load balancing
    - Object k belongs to its forward-nearest machine (its *coordinator*)
    - Also replicates to N successors (its *preference list*) - puts/gets complete if enough quorum count acks
    - *Tunable* consistency level: have applications choose the R/W quorum sizes, s.t.
      - $Q_r + Q_w > N$
      - $2 \cdot Q_w > N$
    - *Sloppy quorum*: only finding the first N live (healthy) nodes
    - *Hinted handoff*: on $W < Q_w$, coordinator can try further nodes, and tells it to periodically try to forward the update back to the intended node
  - Dynamo returns inconsistent results when:
    - During hinted handoff
    - When there is partition
    - When there are multiple failures
- Use *gossiping* for nodes to exchange their list of known nodes - then client connects to any node and at best in one-hop the server is able to redirect the client right to the key's coordinator
- Use *version vectors* (*vector clocks*) (Fig 3.) to automatically resolve some easy-case conflicts - where the service actually can figure out which update is strictly newer, so no need to bother sending both to the client for *reconciliation*

> CAP theorem:
>
> - A + P: Grapevine, Dynamo, ...
> - C + P: Majority protocols (Paxos/Raft/Viewstamp), ...
> - C + A: Chain replication, single-node DB

Maelstrom

Link: https://www.usenix.org/system/files/osdi18-veeraraghavan.pdf

Motivation

- Facebook wants tolerance to datacenter-level physical failures: natural disasters, operational failures, bugs that tear down an entire datacenter
- Challenges:
  1. Heterogeneity of datacenters
  2. Every product is a collection of thousands of services, have complex dependencies
  3. Data or service not deployed in all datacenters due to cost and latency requirements
  4. Continuous growth/change in the products
  5. Fast failover, want no cascading failures

Background

- Facebook's infrastructure design (Fig 1.)

Contribution

- High-level approach in steps:
  1. Global deployment - service/data/storage must be globally deployed to multiple datacenters across the globe
  2. Provision buffer - each datacenter has resilience in its design to accept some exceptional traffic
  3. Replicate (so data is achievable elsewhere if one datacenter fails)
  4. Do traffic redirection (*draining*) upon disaster - focus of this paper
- Recognizes the importance of regular testing, health monitoring, and human intervention
  - Though the dependencies graph is updated by cooperating with the development teams of each product, there might be mistakes, missing/unnecessary dependencies; Hence, do regular *drain tests* to ensure that the dependencies are up-to-date and correct
  - Do *capacity limit monitoring* to decide how much to redirect to each partner datacenter
  - Use the *runbook* UI to allow easy human intervention in case any automation process goes wrong

Distributed File Systems

Distributed file systems are one of the most important application scenario of all the above distributed system theories. They are more challenging than simple KV-stores. Major concerns:

- Load balancing
- Locating data
- Consistent caching
- Recovery & Fault tolerance
- Various sized workloads, access patterns
- Atomicity, file structures, stateful operations
- Access control, permissions

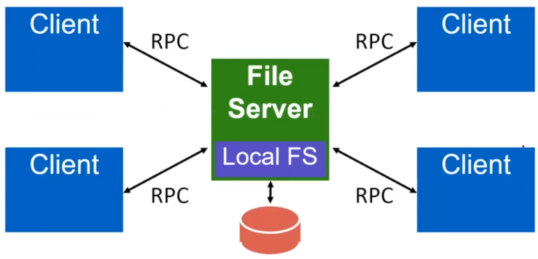
There are typically two flavors of building distributed file systems:

1. Having some central control: NFS, AFS, GFS
2. P2P-flavor, symmetric pool: XFS, Frangipani, Ceph

NFS

Link to the relevant OSTEP chapter: https://pages.cs.wisc.edu/~remzi/OSTEP/dist-nfs.pdf

- Clean server-client model architecture:



- Server exposes its local FS, *blocks of data* cached in client-side *memory*
  - Either allow cache inconsistency to happen, write back on `close()`
  - Or do slow cache coherency protocols

AFS

Link to the relevant OSTEP chapter: https://pages.cs.wisc.edu/~remzi/OSTEP/dist-afs.pdf

- More scalable architecture design
  - Sub directory trees called *volumes* stored across machines
  - Client library knows the global mapping
- Serving whole files on requests, caching of *whole files* in client-side *disks*
  - Cache write back happens if client cached copy changed, and last writer wins
  - *Callback promise* cache invalidation design much like Chubby's `KeepAlive` notifications

GFS

Link: https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

- Classic master metadata server design

XFS

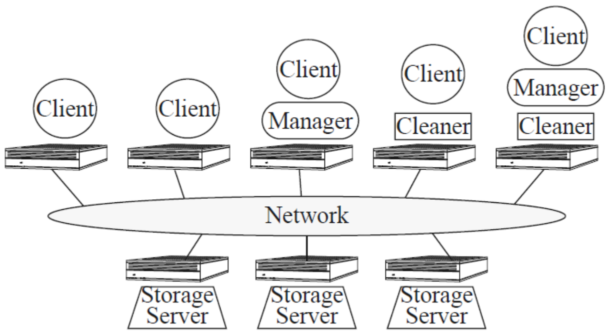
Link: http://lazowska.cs.washington.edu/xfs.pdf

Motivation

- Want a *homogeneous*, *location-independent*, P2P design
  - Breaking a file system into smaller pieces of services
  - Every machine should be able to run any ones of the services

Contribution

- Example of an XFS installation:



  - Nothing at a fixed location; Everything is found via indirection (some map, Tab 1.)
  - Incorporates LFS + RAID striping (Fig 1.); Storage nodes only provide disks, not files
- To locate data on disk:
  - *Manager map*: inode number $\rightarrow$ manager node, globally replicated
  - *Imap*: inode number $\rightarrow$ inode disk log address, split among managers
  - Then find the data block address from inode and find the correct storage server responsible for it through *Stripe group map*
  - Complete file read procedure in Fig 3.
- Techniques to handle load balancing issues in this setting: there will be hot files and hot directories, so a purely uniform data structure placement might get very skewed load on different nodes
  - Have file manager services to decide placement and mapping
  - Tries to put load near client, creates file on/near the client node
  - *Cooperative caching*: if the data block is now cached on some other client, tell the requester to go to that client; Managers remember who is caching which files; Invalidate on write
  - RAID stripe groups, a file could be scattered across multiple machines to allow parallel data transmission
- Each map has its failure recovery scheme (Tab 2.)

Drawbacks

- Incorporates too many things inside one system, not quite a clean design
- No security or authentication mechanisms, but OK

Petal + Frangipani

Link: https://www.scs.stanford.edu/nyu/01fa/sched/petal.pdf

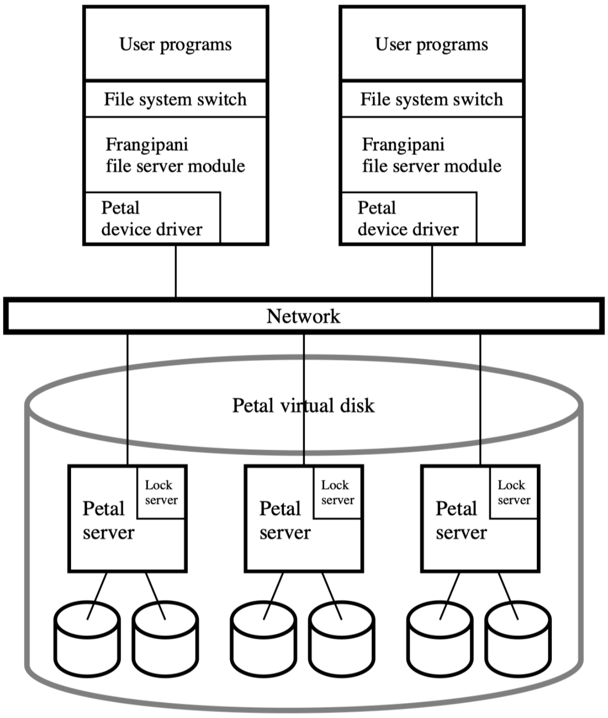
Link: https://pdos.csail.mit.edu/6.824/papers/thekkath-frangipani.pdf

Motivation

- Petal is a distributed *virtual disk* - exposes the block-device level interface
  - Much simpler semantic than distributing a file system, much like KV-store
  - Similar ideas used widely nowadays: AWS EBS, EMC storage, ...
- Shared virtual disk exposes sharing issues & locality issues if used with a tradition UNIX local FS; Frangipani is a file system built upon Petal to make it more powerful

Contribution

- Petal architecture
  - Virtual-to-Physical translation (Fig 4.): `<vdisk, voffset>` → `<server, pdisk, poffset>`
    - *Virtual disk directory* & *Global map* are globally replicated using Paxos
    - *Physical map* is local to each server
    - The global map indirection allows transparent membership reconfiguration (Slide 38., 39.)
    - Client could cache the global map to reduce latency in common case
  - Supports data backup by *copy-on-write* snapshotting; Pauses client operation for app-level consistency
  - It is not very feasible to use parity for disk failure tolerance in a distributed scenario; Petal uses *chained-declustering* (Fig 5.)
    - Each block having two replicas, automatically split traffic to neighbors and then cascadingly propagate
    - Normally, reads could go to either replica (client tracks # pending requests to decide), and writes always first go to primary
  - On write, primary marks data busy (locked), then send the update to both local copy and secondary copy, acknowledging client and clearing busy bit if both complete
    - If unsuccessful, client tries secondary replica
    - If replica detects partner failure, marks data as stale to indicate that partner should re-read them at recovery
- Frangipani design over Petal



  - Assumes a shared disk space, so no leaders, uses lock service (run on Petal servers) for coordination (Fig 2.)
    - Like a Chubby running along with Petal, since Petal uses Paxos anyway
    - Locking is on whole-file granularity and targeting low-sharing workloads
  - Virtual disk space layout (Fig 4.)
    - Large blocks could help with locality given that Petal serves continuous addresses well
    - "Wastes" (Fragments) some of the huge virtual space for efficiency

- For failure recovery, each server has its own FS log somewhere on Petal; Failure of server s detected by the locking mechanism, and any healthy partner could ask for transferring locks and use s's log to recover
- Nice *layering* approach: Frangipani over Petal, both are very simple and clean in design

Ceph

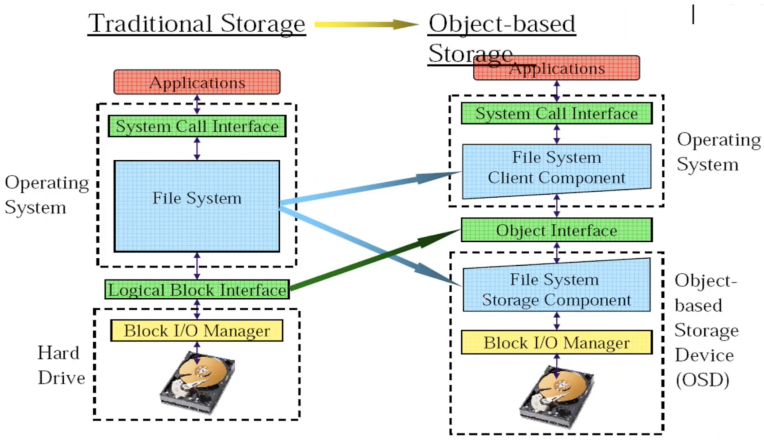
Link: http://www2.cs.uh.edu/~paris/7360/PAPERS07/weil-osdi06.pdf

Motivation

- Seperate data and metadata management, making both sides scalable, not just a single metadata server
- Wants to take the advantage of *object storage* devices

Background

- Object storage devices are smart disks with some of the file system logics offloaded, and providing the upper layer an object ID interface just like a KV-store



Contribution

- Highly-scalable design on both metadata side (MDS) and data side (RADOS of OSDs) (Fig 1.)
  - Gets rid of the file inode mapping completely by using a statically-known hash function called CRUSH from `<file>` $\rightarrow$ `<OSDs, obj>`; Metadata servers hence do not need to replicate these expensive mappings
  - Lookup-based placement (e.g., XFS) vs. Calculation-based placement (e.g., Ceph)
- Data side - RADOS fault-tolerant data distribution with *placement groups* (PGs) and OSDs (Fig 3.)
- Metadata side - globally-known CRUSH hash function (Slide 20.)
  - Deterministic, so known to everyone and no need to dynamically replicate
  - Input does require a hierarchical cluster map of all storage devices and a rule
  - Calculation is done on client, with information returned from MDS
- Metadata side - dynamic directory tree partitioning (Fig 2.) across MDSs for load balancing directory lookup traffic
  - Load- & Locality-aware
  - MDS returns an inode number to client, client then does CRUSH

Authentication, Privacy, & Security

Kerberos

Link: https://www3.nd.edu/~dthain/courses/cse66771/summer2014/papers/kerberos.pdf

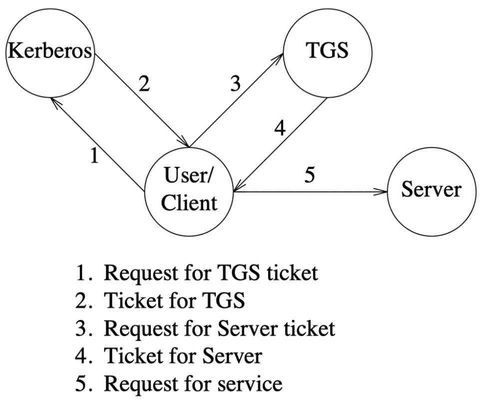
Motivation

- *Authentication* == Prove to a server over a network who you are. Previous solutions have exposed vulnerabilities:
  - `rlogin` - minimum authentication, just check username + IP address
    - Trust all client OS to provide correct username (logged in on client machine)
    - Easy to do *man-in-the-middle* (MITM) attack
  - `ftp` - server has database of users & passwords, client sends username & password
    - Attackers could do MITM network sniffing to learn about the passwords
  - *Challenge* mechanism

    - To scale up, could use a separate *domain controller* (DC) entity to store the database; Server talks with DC and DC replies yes/no
    - MITM still easy, just relay all the messages
- Want a scalable & decoupled authentication system: client OS no longer trusted, any untrusted client laptop (no just workstations in a fixed network) should be able to access services/data

Contribution

- Defines a clear threat model
  - Threats:
    - Network sniffing
    - Record/Replay attacks
    - Client OS compromised
  - Constraints (Goals):
    - Multiple services using Kerberos, each with their own access rules
    - IP spoofing is hard
    - Scalable and highly available
    - *Single sign-on*: user login once and have access to all university services
    - Separate database from auth server
    - Don't store password on client
- Kerberos essential design & workflow:
  - Participants:
    - Client c
    - Kerberos server == the authentication server (database check not shown in this figure)
    - TGS tgs == the *ticket-granting server*
    - Server s == the service that uses Kerberos for authentication
    - K_x means the private key owned by entity x
    - $K_{x,y}$ means a random session key with expiration lifetime for communication between x and y
  - Kerberos *Ticket-granting Ticket* for single sign-on: $T_{c,tgs} = tgs, c, addr, timestamp, lifetime, K_{c,tgs}$
  - Kerberos *Service Ticket* for finally talking with the service s: $T_{c,s}$ in similar structure
  - Kerberos *Authenticator* constructed once per service access: $A_c = c, addr, timestamp$
  - Authentication workflow:



    1. c, flag saying wants tgs
    2. $\{K_{c,tgs}, \{T_{c,tgs}\}_{K_{tgs}}\}_{K_c}$

       These two are single sign-on operations.

       Why need TGS? We want to store something more secure than plain password on the client.
    3. $s, \{A_c\}_{K_{c,tgs}}, \{T_{c,tgs}\}_{K_{tgs}}$

       Client OS asked for password, then use it to decrypt 2.. Client constructs A_c.
    4. $\{K_{c,s}, \{T_{c,s}\}_{K_s}\}_{K_{c,tgs}}$

5. $\{A_c\}_{K_{c,s}}, \{T_{c,s}\}_{K_s}$
6. (optional if the client wants mutual authentication) server sends back $\{timestamp + 1\}_{K_{c,s}}$

- Guards against replay attacks by having timestamps, lifetimes, and *replay cache* on servers that rejects all replayed messages seen in the last 5 minutes
- Used widely in universities and inspired nowadays popular *OpenID* systems, e.g., "Sign in with Google", ...

Get Off My Cloud

Link: https://hovav.net/ucsd/dist/cloudsec.pdf

Background

- Cloud computing gets popular
  - *Software-as-a-Service* (SaaS): e.g., Google docs
  - *Infrastructure-as-a-Service* (IaaS): provide virtual machines and networking to customers
  - *Platform-as-a-Service* (PaaS): IaaS but with OS support and common services taken care of
  - *Function-as-a-Service* (FaaS): i.e., *serverless computing*, see the last section
- Threats to cloud computing security:
  - Multi-tenancy, VM co-location (co-residency) + side-channel leaks
  - Trust cloud provider and virtualization software
  - Performance interference

Contribution

- Describes a practical information leak attack on AWS
  - Learn the VM allocation policy and establish a mapping
    - Binpacking algorithm of AWS for locality
  - Use this knowledge to try to achieve VM co-residency with target victim, and do co-residency checks to be sure
  - Exploit various covert/side-channel attacks on shared physical resources to leak information about the victim, e.g.,
    - Disk seek timing for stronger co-residency checks
    - Leak performance stats
    - Keystroke timing for guessing passwords
    - Cache side-channel attacks

Zanzibar

Link: https://www.usenix.org/system/files/atc19-pang.pdf

- Google's distributed ACL service

Serverless Computing

Serverless Workloads

Link: https://www.usenix.org/system/files/atc20-shahrad.pdf

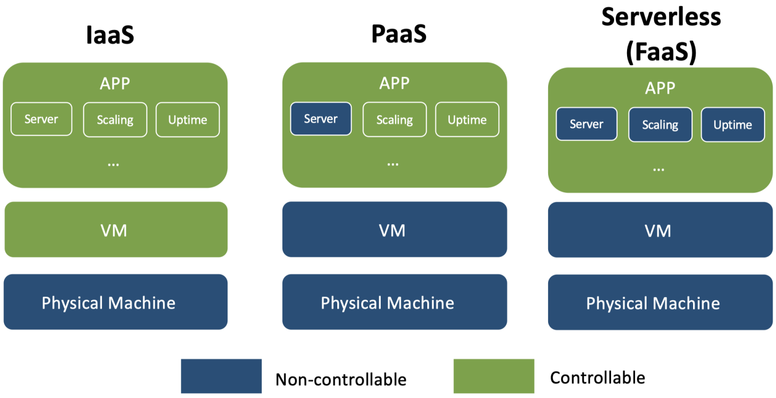
- Various workloads have different impacts on the *cold start* issue

Peeking Serverless

Link: https://www.usenix.org/system/files/conference/atc18/atc18-wang-liang.pdf

Background

- Serverless functions (FaaS) have the following advantages compared to IaaS:
  - Scaling down when not needed (elasticity)
  - Scaling up very quickly
  - Quick response time and deployment cost
  - Better programmability, only need to care about the actual app logic
  - Cost-efficiency, fine-grained billing scheme



Contribution

- Comprehensive empirical study on three serverless platform providers: AWS Lambda, MS Azure Functions, & Google Cloud Functions
  - Using VM vs. Docker vs. Processes; Isolation level, at which level is multi-tenancy (Fig 2.)
  - Cold start launching performance (Slide 31., Fig 8.)
  - Instance lifetime (Slide 39.)
  - Instance placement & contention (Slide 34., 37.)
  - Function resource configurations: CPU, memory, storage (Slide 36.)