OS: User interface to hardware (device driver), Provides abstractions (processes, file system), Resource manager (scheduler),Means of communication (networking)
Distributed systems: client-server(NFS), Web, internet, A wireless network, DNS, cloud, dc
Cloud = Lots of storage + compute cycles nearby
WUE = Annual Water Usage / IT Equipment Energy
PUE = Total facility Power / IT Equipment Power low is good
4 features in cloud: Massive scale, On-demand access, Data-intensive Nature, New Cloud Programming Paradigms
HaaS: Hardware as a Service
IaaS: Infrastructure as a Service (AWS)
PaaS: Platform as a Service (Google's AppEngine)
SaaS: Software as a Service
FOLDOC errors: one machine(web), client-server(p2p like bitT)
A **distributed system** is a collection of entities, each of which is autonomous, programmable, asynchronous and failure prone, and which communicate through an unreliable
communication medium.
YARN Scheduler: Treats each server as a collection of containers, has 3 components:
– Global Resource Manager(**RM**): Scheduling; Per-server Node Manager(**NM**): Daemon and server-specific functions; Per-application (job) Application Master(**AM**): Container negotiation with RM and NMs, Detecting task failures of that job
Speculative Execution:Perform proactive backup (replicated) execution of some straggler tasks
Locality:rack-fault-tolerance:2 on a rack, 1 on a different rac; Mapreduce attempts to schedule a map task on:
1. a machine that contains a replica of corresponding input data, or failing that, 2. on the same rack as a machine containing the input, or failing that, 3. Anywhere
Tree-based multicast: O(N) ACK/NAK overhead
Gossip: pull gossip is faster than push gossip, Second half of pull gossip finishes in time O(log(log(N))
Topology-Aware Gossip: Random gossip target selection => core routers face O(N) load; in subnet, Router load=O(1)
Failure Detectors: Desirable Properties: speed, scale,
• **Completeness** = each failure is detected (guarantee)
• **Accuracy** = there is no mistaken detection (partial)
(**Impossible together in lossy networks)
Gossip-Style Failure Detection:
•Nodes periodically gossip their membership list: pick random nodes, send it list
•On receipt, it is merged with local membership list
•When an entry times out, member is marked as failed
*If the heartbeat has not increased for more than Tfail seconds, the member is considered failed
*And after a further Tcleanup seconds, it will delete the member from the list (if not this, other processes may not have deleted that entry and it may be added bac, ping-pong)
*N heartbeats, O(log(N)) time to propagate, if bandwidth allowed per node is allowed to be O(N), O(Nlog(N)) time to propagate, if bandwidth allowed per node is only O(1)
SWIM Failure Detector: Completeness: Any alive member detects failure Eventually. 2n-1. Suspicion with incarnation.
Dissemination Options: Multicast, Point-to-point, Piggyback
P2P: **Napster**: Server stores no files.TCP. ternary tree. Query, search, response, ping candidates, download from best host. Probs: Congestion, single point of failure, no security
Gnutella: no servers, clients as servants. overlay graph.
Search: Query's flooded out, ttl-restricted, forwarded (to all neighbors) only once, Successful results QueryHit's routed on reverse path. If firewall, push, tcp, get. Periodic Ping-pong to continuously refresh neighbor lists.
Probs: Ping/Pong 50% traffic, Repeated searches(cache query), bw(central server as proxy), freeloaders, flooding.
FastTrack: Hybrid between Gnutella and Napster, "healthier" participants(supernodes), provided earned enough reputation. A peer searches by contacting a nearby supernode
BitTorrent: Download Local Rarest First block policy: early download of blocks that are least replicated among neighbors
Tit for tat bw usage: Provide blocks to neighbors that
provided it the best download rates; Choking: Limit number of neighbors to which concurrent uploads <= a number (5)
DHT: Napster(O(1)), Gnutella(O(N)), FastTrack, Chord(O(log(N)))
Chord: id >= (n+2i) mod 2m; Consistent Hashing => with K keys and N peers, each peer stores O(K/N) keys.failure: maintain r (r=2log(N)) multiple successor entries. In case of failure, use successor entries; replicate file/key at r successors and predecessors
Churn Rate: Hourly peer turnover rate
Stabilization protocol: Introducer directs N40 to N45 (and N32), N32 updates successor to N40, N40 initializes successor to N45, and inits fingers from it N40 periodically talks to neighbors to update finger table(concurrent might -> loops)
Number of messages per peer join=O(log(N)*log(N))
Hash can get **non-uniform** -> Bad load balancing (sol: Treat each node as multiple virtual nodes behaving independently)
Pastry: Assigns ids to nodes, just like Chord (using a
virtual ring); **Leaf Set** - Each node knows its successor(s) and predecessor(s)
Routing tables based on **prefix matching**, log(N)
route to a peer, starts by forwarding to a neighbor with the largest

Consistent Cut: for (each pair of events e, f in the system)
event e is in the cut C, and if f → e, then: f is also in the cut C
(can't both) Liveness = guarantee that something good will happen, eventually; Safety = guarantee that something bad will never happen
FIFO Ordering: msgs sent from the sender same order as they delivered
Send multicast at process Pj:
– Set Pj[j] = Pj[j] + 1
– Include new Pj[j] in multicast message as
its sequence number
• Receive multicast: If Pi receives a multicast
from Pj with sequence number S in message
– if (S == Pi[j] + 1) then
• deliver message to application
• Set Pi[j] = Pi[j] + 1
– else buffer this multicast until above
condition is true
Causal Ordering: FIFO < Causal, M3:1 -> M3:2 / M1:1 -> M3:1, and so should be received in that order at each receiver
• Send multicast at process Pj:
– Set Pj[j] = Pj[j] + 1
– Include new entire vector Pj[1…N] in multicast message as its sequence number
• Receive multicast: If Pi receives a multicast from Pj with vector
M[1…N] (= Pj[1…N]) in message, buffer it until both:
1. This message is the next one Pi is expecting from Pj, i.e.,
• M[j] = Pi[j] + 1
2. All multicasts, anywhere in the group, which happened-before M have been received at Pi, i.e.,
• For all k ≠ j: M[k] ≤ Pi[k]
• i.e., Receiver satisfies causality
3. When above two conditions satisfied, deliver M to application and set Pi[j] = M[j]
Total Ordering: all receivers receive all multicasts in the same order
Reliable multicast loosely says that every correct process in the group receives all multicasts
Virtual Synchrony: The set of multicasts delivered in a given view is the same set at all correct processes that were in that view. No reliably delivered also does not sat. View has to include the sender
Consensus problem: agreement. all processes decide all 0s or all 1s (FLP - impossible in async model) constraints: validity(if everyone proposes same val, that's what's decided), integrity(decided val must have been proposed by some procs), non-triviality(exists >= 1 init system state that leads to all 0s/1s).
Paxos: does not solve consensus but safety and eventual liveness
has rounds; each round has a unique ballot id;
Phase 1: election: Potential leader chooses a unique ballot id, sends to all, quorum respond OK then you are the leader (cannot 2 leaders)
Phase 2: Proposal, v OK?; Phase 3: Decision, v!
Leader Election: Safety: For all non-faulty processes p: (p
's elected = (q: a particular non-faulty process with the best attribute value) or Null); Liveness: election run terminates
Ring: worst case: N-1+N+N=3N-1 msgs; best case: 2N msgs
Bully: send it to process that have a higher id than itself. if no one answers, call itself leader. When receives election message, replies OK, and starts its own election.
If failures stop, eventually will elect a leader
Worst-case: 5 message transmission times; best-case: 1
Election in **chubby**: server with majority of voters becomes new leader, every node votes for at most one. relies on paxos-like protocol, can also write small configuration files. safe & eventual live.
Mutual Exclusion:
• Safety (essential):At most one process executes in CS at any time
• Liveness (essential):Every request for a CS is granted eventually
• Ordering (desirable):Rqts are granted in the order they were made
Cental Solution: safe, live, FIFO ordering, bw:2 enter, 1 exit, client delay:2; sync:2; bottleneck, SPoF
Ring-based Mutex: pass token to successor. Safe, live, bw: 1~N enter, 1 exit; client delay: 0~N; sync: 1~N-1
Ricart-Agrawala: safe, live(N-1 worst case), bw(bad!): 2(N-1) enter, n-1 unicast(1 multi) exit, client delay: 1 rtt, sync(good): 1
• enter() at process Pi
• set state to Wanted
• multicast "Request" <Ti, Pi> to all processes, where Ti =
current Lamport timestamp at Pi
• wait until all processes send back "Reply"
• change state to Held and enter the CS
• On receipt of a Request <Tj, Pj> at Pi (i ≠ j):
• if (state = Held) or (state = Wanted & (Ti, i) < (Tj, j))
// lexicographic ordering in (Tj, Pj)
add request to local queue (of waiting requests)
else send "Reply" to Pj
• exit() at process Pi
• change state to Released and "Reply" to all queued requests.
Maekawa's: get replies from some procs. Voting set (quorum), size k, each proc belongs to M other Vs. k=M=sqrt(n) best. Take the row and column it belongs to. Safe, not live!(deadlock, loop), bw: 2sqrt(n) enter, sqrt(n) exit, client delay: 1 rtt; sync: 2
Remote Procedure Call: at most(Java RMI)/least once(Sun RPC)(hard), maybe(CORBA)(weakest)
LPC: exactly-once semantics, If process is alive, called function executed exactly once
Idempotent op: can be repeated mult. times w/o side effects.
Marshalling: caller xx -> CDR (common data rep, platform ind.)
Unmarshalling: callee CDR -> xx
Retvals marshalled on callee and unmarshalled on caller

matching prefix, 011101*; among all potential
neighbors with the matching prefix, the neighbor with the shortest round-trip-time is selected.
early hops are short and later hops are longer
Kelips – A 1 hop Lookup DHT, k affinity groups, hash mod k
Affinity group does not store files. PennyLane.mp3 hashes to k-1,Everyone in this group stores <PennyLane.mp3, who-has-file>
NoSQL: get(key) and put(key, value)
Unstructured, Columns Missing from some Rows, No schema, No foreign keys, joins may not be supported; Col-oriented(Range searches within a column are fast)
***Cassandra**: Ring-based DHT without finger tables
Replication Strategy: two options:
1.SimpleStrategy:random partitioner, chord like hashing; Byte Order Partitioner: Assign ranges of key to server
2.NetworkTopologyStrategy: Several Replicas Per DC; for every DC, first replica placed according to partitioner, go clockwise until hit a different rack
Snitches: SimpleSnitch, PropertyFileSnitch, EC2Snitch,
Rack Inferring: x. <DC octet>. <rack octet>. <node octet>
Writes: lock-free & fast
Always writable: Hinted Handoff mechanism:
-any replica down, coordinator writes to all other replicas, and keeps the write locally until down replica comes back up.
-all replicas are down, the Coordinator buffers writes
One ring per datacenter
Receiving writes: memtable: append only
SSTable(SortedStringTable): immutable(once created,don't change)
Bloom Filter: Low FP=$(1-(1-1/m)^{kn})^k$, m-#bits, k-#functions, n-#items, Never false negatives
On check-if-present, return true if all hashed bits set(cheap).
Deleting: add tombstone, delete when compaction
Cassandra uses gossip-based **cluster membership**:
Suspicion:PHI(t)=-log(CDF or Probability(t_now – t_last))/log 10
CAP Thm: In a distributed system you can satisfy at most 2 out of the 3 guarantees: 1. Consistency: all nodes see same data at any time, or reads return latest written value by any client 2. Availability: the system allows operations all the time, and operations return quickly 3. Partition-tolerance: the system continues to work in spite of network partitions
-Cassandra, Riak, Dynamo, Voldemort: Eventual (weak) consistency, Availability, Partition-tolerance; RDBMSs: Strong consistency over availability under a partition; HBase,spanner: Consistency, fault-tolerance over avail
Eventual Consistency: If all writes stop (to a key), then all its values (replicas) will converge eventually.
-RDBMS provide **ACID**: **Atomicity**(All or nothing),**Consistency**(consistent state),**Isolation**(atomic btw transacs), **Durability**(transac complete->saved)
-Key-value stores like Cassandra provide **BASE** (Basically Available Soft-state Eventual Consistency)
consistency levels: ALL<QUORUM(strong consistency)<ONE<ANY
Quorums: W + R > N, W > N/2 (R/W = read/write replica count) (W=1, R=1): very few writes and reads; (W=N, R=1): great for read-heavy workloads; (W=N/2+1, R=N/2+1): great for write-heavy
Workloads; (W=1, R=N): great for write-heavy workloads
with mostly one client writing per key
HBase:Zookeepers, small group of servers running Zab (Paxos-like consensus protocol). Regions, memstore, hfile
Consistency models: Eventual < causal < per-key seq. < red-blue < probabilistic < CRDT < Seq. < Linearizability
Time synchronization: Correctness & fairness
Clock Skew = Δclock values of two processes (distance)
Clock Drift = Δclock frequencies/rates of two processes (speed)
Max drift rate between two clocks with similar MDR is 2*MDR
Given a maximum acceptable skew M between, need to synchronize at least once every: M / (2 * MDR)
External(|C(i) – S| < D): Cristan's Alg: [t + min2, t + RTT - min1], t P set time to t + (RTT+min2-min1)/2,error at most (RTT-min2-min1)/2
NTP: Offset o = (tr1 – tr2 + ts2 – ts1)/2
Internal(|C(i) – C(j)| < D): Lamport: ts = max(local, msg) + 1
E1 < E2 ⇒ timestamp(E1) < timestamp (E2), BUT
timestamp(E1) < timestamp (E2) ⇒ {E1 < E2} OR {E1 and E2 concurrent}
Vector: Vi[i] = Vi[i] + 1, Vi[j] = max(Vmessage[j], Vi[j]) for j ≠ i
Two events are causally related iff VT1 < VT2, i.e.,iff VT1 ≤ VT2 & there exists j s.t. 1 ≤ j ≤ N & VT1[j] < VT2[j]
Chandy-Lamport Global Snapshot Alg:
*First, Initiator Pi records its own state
• Initiator process creates special messages called "Marker" messages
• for j=1 to N except i, Pi sends out a Marker message on outgoing channel Cij, total (N-1) channels
• Starts recording the incoming messages on each of the incoming channels at Pi: Cji (for j=1 to N except i)
*Whenever a process Pi receives a Marker on an incoming channel Cki
• if (this is the first Marker Pi is seeing)
– Pi records its own state first
– Marks the state of channel Cki as "empty"
– for j=1 to N except i
• Pi sends out a Marker message on outgoing channel Cij
– Starts recording the incoming messages on each of the incoming channels at Pi: Cji (for j=1 to N except i and k)
• else // already seen a Marker message
– Mark the state of channel Cki as all the messages that have arrived on it since recording was turned on for Cki

Serial Equivalence: all conflicting pairs are executed in the same order
Pessimistic concurrency control: locking
Optimistic: Check at commit time, multi-version approaches
2-phase locking: a T cannot acquire/promote any locks after it starts releasing locks. Strict: only releases at commit pt. Downside:deadlock!
3 conds for deadlock: some objs are accessed in exclusive lock modes, locks on hold cannot be preempted, circle wait
First-cut:check for SE at commit => cascading aborts :(
Timestamp Ordering: T's write(read) to object O allowed only if transactions that have read or written(written) O had lower ids than T.
Multi-version Concurrency control(MVCC): tentative+committed ver.
eventual consistency in Cassandra and Dynamo DB: L**ast write wins,** timestamp based on physical time
eventual consistency in Riak: vector clocks, size-based / time-based pruning
Stream Processing: Grouping: shuffle grouping (round robin), fields grouping (group a stream by a subset of its fields), all grouping (all tasks of bolt receive all input tuples)
Master + Worker + Zookeeper
Twitter's Heron System:in combination of the 3 following methods:
-TCP Backpressure -Sprout Backpressure -Step by Step Backpressure
Graph Processing: Google's Pregel System (Master + Worker)
Master assigns a partition of vertices to each worker. Run iteration until no vertices are active and no messages in transit
Replication Control: nines availability, transparency(clients not aware of mult.), consistency(all clients see same data)
Passive Replication: uses a primary replica (master)
Active Replication: treats all replicas identically
One-copy serializability: concurrent transac == serial transac, in ono-replicated system, correctness == serial equivalence
2-phase commit: all yes or abort.
Structure of Networks: magic # 6, small world nw(high CC, short paths)
Clustering Coefficient = P(A-B given A-C & C-B)
Power Law: $k^{-\alpha}$ scale free network, linear in log(#degree,k) and log(#nodes)
Both power law and small world nw: internet backbone, telephone call graph, www, gnutella.. Resilience, routing overhead on high-degree vertices
Scheduling: FIFO(avg completion time high), Shortest Task First(optimal)(batch), RR(interactive)
-**Hadoop Capacity Scheduler** (multiple queues, can have hierarchy, FIFO in each queue, elastic(allowed to occupy more clusters if rsc free), preempt not allowed)
-**Hadoop Fair Scheduler** (Divides cluster into pools (Typically one pool per user), Resources divided equally among pools), When minimum share not met in a pool, Pre-emption occurs, To kill, scheduler picks most-recently-started tasks
Dominant-Resource Fair Scheduling: tenant can't benefit from lying, envy-free: can't envy other tenant's allocations
Job1<2 cpus, 8 gb>, job2<6 cpus, 2 gb>, cloud total<18 cpus, 36>
Calculate dominant rsc: job1: cpu=2/18, ram=8/36 => mem-intensive, job2: cpu=6/18, ram=2/36 => cpu-intensive. => job1 ram% = job2 cpu%
Rule: dominant resources same for all jobs.
Concurrent Accesses: One-copy update semantics: when file is replicated, its contents, as visible to clients, are no different from when the file has exactly 1 replica.
Vanilla DFS: Flat File Service API
- No automatic read-write pointer! Need operation to be idempotent (at least once semantics)
- No file descriptors! Need servers to be stateless: easier to recover after failures (no state to restore!)
Server Optimizations: server caching(store recently accessed blks), locality, writes 2 flavors: 1.delayed: write in mem, fast not consistent; 2.write-thru: write in dick imm, consistent but slow.
Client caching: each blk in cache tagged w/ Tc(last time validated), Tm(last time modified), a cache entry valid if T-Tc<t or Tm_client = Tm_server, t(freshness interval), when blk written, do a delayed write to server
Andrew File System (AFS): optimistic R/W, Callback promise:Promise that if another client modifies then closes the file, a callback will be sent from Vice (Server) to Venus (Client)
Two unusual design principles: Whole file serving, Not in blocks; Whole file caching, Permanent cache, survives reboots
Based on (validated) assumptions that: Most file accesses are by a single user; Most files are small; Even a client cache as "large" as 100MB is supportable
Invalidate Protocol: downsides: flip-flop 2 procs write same page; lots of network transfer; false sharing(unrelated vars fall in same page, when page large)
Update Protocol: mult. Procs can W state. On write, multicast newly written vals to all other W holders. Prefered when lots of sharing among procs, writes are small vars, page large.
Sensors: canonical sensor node contains: converter of energy form, microprocessor, comm link, power src.
TinyOS: Event-driven execution (reactive mote), Modular structure (components) and clean interfaces, Static allocation only avoids run-time overhead, Scheduling: dynamic, hard (or soft) real-time, explicit interfaces
in-network aggregation approaches: Build trees among sensor nodes, base station at root of tree, Internal nodes receive values from children, calculate summaries (e.g., averages) and transmit these, More power-efficient than transmitting raw values or communicating directly with base station
Chandy-Lamport Global Snapshot Alg: (cont)
*The algorithm terminates when
• All processes have received a Marker
– To record their own state
• All processes have received a Marker on all the
(N-1) incoming channels at each
– To record the state of all channels