Cloud = storage + compute cycles nearby.
Clouds' Feature: [Massive scale, On-demand access [HaaS IaaS PaaS SaaS], Data-intensive nature, New cloud programming paradigms].
WUE = annual water usage / IT equipment energy.
PUE = total facility power / IT equipment power.

YARN Scheduler:
[Global **R**esource **M**anager (scheduling),
Per-server **N**ode **M**anager (daemon; server-specific),
Per-app/job **A**pplication **M**aster (container negotiation with RM and NMs; detecting task failures of the job)]
{AM needs container to RM; NM notify RM container available; RM notify AM location; AM req NM task}
Server Failure: {NM heartbeats RM, RM notify all affected AM; NM keep track of its tasks, mark as idle and restart; AM heartbeats RM, RM restart AM, sync}
RM Failure: {Use old checkpoints, bring up 2nd RM}
Heartbeats piggyback container requests

Spanning Tree based Multicast
1. **S**calable **R**eliable **M**ulticast: {NAK; add random delay, exponential backoff to avoid storms}
2. **R**eliable **M**ulticast **T**ransport **P**rotocol: {ACK}
O(N) ACK/NAK overhead

Population: n+1, Uninfected: x, Infected: y.

$$x = \frac{n(n+1)}{n + e^{\beta(n+1)t}}, y = \frac{(n+1)}{1 + ne^{-\beta(n+1)t}}$$

t = clog(n), y = n+1-(1/n^{cb-2}), each node < cblogn msgs.

Pull Gossip: first half = O(log(N)) rounds, second = O(log(log(N))) time.

Heartbeating: {Centralized, Ring, All-to-All: [L = N/T], Gossip: [T = logN * T_{gossip}, L = N/T_{gossip}]}
Gossip period T ↑: detection time ↓, false rate ↑.
SWIM: E(T) = [1-(1-1/N)$^{N-1}$] * (e/e-1)
Suspicion Mechanism: incarnation number per process

P2P SYSTEMS:
1. Napster: [TCP]
Server stores no files; maintains <file, ip, port> list.
Client pings each host @ list, fetch from fastest.
2. Gnutella:
Servents connected in an overlay graph.
[Query; QueryHit; Ping; Pong; Push]
HTTP: standard; well-debugged; widely used.
Avoid excessive traffic:
Peer maintains recently received msg list.
Forward to neighbors except sender.
Each Query forwarded only once.
Duplicates (DescriptorID and msg type) dropped.
QueryHit's DscptID for unseen Query dropped.
Problems:
Ping/Pong take 50% traffic → Multiplex, cache.
Repeated serach → Cache Query & QueryHit.
Modem-hosts small bandwidth → central server act as proxy.
3. Fasttrack (Kazza, Kazzalite, Grokster):
Supernode stores a dir listing a subset <file, peer>.
Supernode membership changes over time: enough reputation. Save metadata locally and query fast.
4. BitTorrent: {Get tracker; Get peers; Get file blocks {Download Local Rarest First block; Tit for tat: 投桃报李; Choking (limit number to best neighbors)};}.

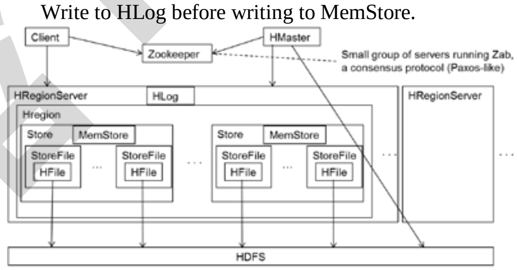
P2P SYSTEMS W/ PROVABLE PROPERTIES:
1. Chord:
<u>D</u>istributed <u>H</u>ash <u>T</u>able [load balancing, fault tolerance, efficiency of lookups and inserts, locality]
Consistent Hashing [SHA-1 (160 bit); truncate to m bits; peer id 0 to 2^m-1]
Peer failure: replicate file/key at r suc/prede cessors.
2. Pastry:
Prefix matching.
3. Kelips:
k (~ √N) affinilty groups, each node hash to one.
All nodes @ group replicate pointer <file, location>.
Membership list dissemination time O(log(N)).

<u>N</u>ot <u>only</u> <u>SQL</u>: [get(k), put(k, v); ~~join?, foreign keys?~~]
{Tables: Column family @ Cassandra, Table @ HBase, Collection @ MongoDB; Column-oriented storage}
1. **Cassandra** [write faster than read]
DHT without finger tables or routing.
Partitioner: Key → server mapping.
Client → Coordinator (any server) → Replicas.
Replication strategy:
a. Simple Strategy
RandomPartitioner (Chord-like hash),
ByteOrderedPartitioner (range query).
b. Network Topology Strategy
For multi-DC deployments.
Snitches: x.<DC>.<rack>.<node>
Write: {Coordinator use Partitioner to send query to all replicas responsible, returns ack, {Memtable [append-only, last write wins], <u>S</u>orted <u>S</u>tring <u>Table</u> [immutable], Bloom Filter}}
Delete: Add tombstone to log.
Read: Coordinator contact **X** replicas in same rack & fetch from other replicas. (init. Read Repair if diff.);
Replica looks at Memtable first, then SSTables.

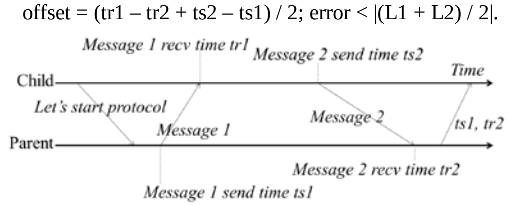
CAP Theorem [Consistency, Availability, Partition-Tolerance] *RDBMS: Atomi Consis Isola Durabi*
X: W+R > N; W > N/2.

W	R	Situation
1	1	Few writes & reads
N	1	Read-heavy workloads
N/2+1	N/2+1	Write-heavy workloads
1	N	Write-heavy & 每 key 1 client W

Linearizability > Sequential (Lamport) > Commutative Replicated Data Types > Red-Blue > Causal > Eventual
HBase [strong consistency + partition-tolerance]
Table: 横 Region, 纵 ColumnFamily.
Write to HLog before writing to MemStore.



Clock Skew: relative diff in value; **Clock Drift**: relative diff in clock freq/rate; Sync at least every M / (2 * <u>M</u>aximum <u>D</u>rift <u>R</u>ate).
Christian's Algorithm: {P→S: min1, S→P: min2; t = [t + min2, t + RTT – min1]; err < (RTT-min2+min1)/2}
Network Time Protocol: organized in a tree.
offset = (tr1 – tr2 + ts2 – ts1) / 2; error < |(L1 + L2) / 2|.



Lamport Timestamp: @instr: add 1; @send: add 1; @receive: max(local, msg + 1).
E1 → E2 ⇒ LT(E1) < LT(E2);
LT(E1) < LT(E2) ⇒ {E1 → E2} **OR** {E1 || E2}.

Vector Timestamp:
Causally related iff. $VT_1 < VT_2$
$VT_1 \parallel VT_2$ = NOT $VT_1 \le VT_2$ *AND* NOT $VT_2 \le VT_1$

Global Snapshot / Global State = individual state of each process + each communication channel.
Snapshot should not interfere with normal app action or require app to stop sending msgs.

Chandy-Lamport Global Snapshot Algorithm:
{Initiator Pi <u>records own state, sends Marker on outgoing channel Cij, starts recording incoming msg on these channels</u>; When Pi receives Marker: {if not seen before: do the underlining actions, mark Cij as empty; if seen: mark 仅 Cji as all msgs arrived on it.}; Terminate when all P receives Marker / on all incoming channels.**}**
Consistent Cut: if e in cut, then all the f→e also in cut.

Liveness: something good will eventually happen.
Safety: anything bad will never happen.

MULTICAST ORDERING
1. FIFO Ordering:
Multicasts 接收的和发送的顺序一样, per sender.
{Pi maintains Pi[1…N]; Send @ Pj, Pj[j]++, include Pj[j] in msg; Receive Pj's S @ Pi: {if S==Pi[j] + 1: deliver, 仅 Pi[j]++; else: buffer until condition true}}
2. Causal Ordering:
Causally related 的 multicast 接收时仍符合因果.
Causal Ordering (更强) → FIFO Ordering (更弱)!
{Pi maintains Pi[1…N]; send @ Pj: Pj[j]++, include entire vector; receive M[] @ Pi , buffer until it is the next one Pi is expecting from Pj AND all msg before M is received (i.e. for k≠j: M[k]≤Pi[k])**}**
3. Total Ordering (Atomic Broadcast):
Every process 收到所有 multicasts 的顺序一致.
{Sequencer: global sequence number S = 0, local received sequence Si = 0; receives @ Sequencer M: set S++, multicast <M, S>; receives @ Pi: buffer until receives <M, S(M)> AND Si + 1 == S(M)**}**
Virtual Synchrony / View Synchrony: {each process maintains a membership list called View, its update called View Change; VS guarantees: All View Changes are delivered in the same order at all correct P; M is "delivered in view V at process P" if P receives V, then BEFORE P receives next view, it delivers M.}

CONSENSUS [Safety (protocol doesn't end with nodes in disagreement) & Liveness (protocol ends)]
1. Synchronous System Model [msg rcved in bounded time; bound clock drift per P; lb & ub each step per P]
Consensus at most f crashing: {$Values^r_i$: proposed values set known to Pi at beginning of round r; Initially $Values^0_i$ = {}, $Values^1_i$ ={vi}; For rounds 1 to f+1, multicast $Values^r_i$ - $Values^{r-1}_i$, $Values^{r+1}_i$ = $Values^r_i$, for each Vj received: $Values^{r+1}_i$ ∪ Vj; d_i = min($Values^{f+1}_i$) }

2. Asynchronous Sys Model: Consensus impossible.

STATE: Configuration = global collection state for each P. **Event** [atomic] {receipt of msg; proc of msg; sending out msg}. **Schedule** (sequence of events)

Easier Consensus Problem: some P eventually set to 0/1
FLP: {exists initial a bivalent conf.; starting from a bivalent conf. always reach another bivalent conf.}

Paxos [*Safety + eventual liveness*] {Async rounds, each a unique ballot id; Phases {Election; Bill; Law}}
{P restarts: uses log to retrieve a past decision and past-seen ballot ids; Leader fails: starts new round; Anyone can start a round any time, may never end}

ELECTION [Safety: all non-faulty p: elect best value q or null; Liveness: all runs: {run terminates AND all non-faulty p: elect a non-null}]

1. Ring Election [logical ring ≈ Chord; Worst 3N-1, best 2N] {Initiator: any Pi initiates an "Election" on finding old coordinator / leader fails; Pi rcv "Election": {if its attr > own: forward it; if < AND Pi hasn't forward one; overwrite w/ own, forward; else if =: Pi becomes new coordinator, sends "Elected"}}
Liveness violated if best one crashes after sending.

2. Bully Algorithm [All P know other P's IDs] **{**When Pi finds coordinator fails: {if it is the max: elects itself, sends "Coordinator" msg to all; else: sends "Election" to all Pj with higher IDs.}; After election starts, {if Pi receives no reply in timeout: elects itself, sends out "Coor"; else: wait for "Coor", starts new if no reply}; Pi receives "Election": replies "OK", starts own election;**}**

[Async: Liveness not guaranteed; Sync: satisfy] [Eventually elect one if failures stop; Timeout length: 5 or 4 msg transmission time (Lowest's Election + (2^{nd}'s answer to it || 2^{nd}'s Election to 1^{st}) + 2^{nd} timeouts + Coor from 2^{nd}); #Msg [Worst nC2 = $O(N^2)$; best N-2]]

MUTUAL EXCLUSION [Safe; Liveness; Ordering?]
Sys Model [TCP; Msg eventually FIFO; P not fail]
Critical Section [≤1] {Enter(); AccessRes(); Exit()}

1. Semaphores [S 最大允许] {Wait/P/Down(S): 阻塞直到 S > 0, enter(); Signal/V/Up(S): exit()}

2. Central Solution [Safety, eventual liveness, FIFO] {Elect a central master/leader, who keeps a queue of waiting reqs and a special token; Any P: {enter(): send req to master, wait token; exit(): send back token to master;} Rcv Pi's req @ Master: {if has token: send to Pi; else: add Pi to queue}; Rcv Pi's token @ Master: {if queue empty: keep; else: dequeue head Pj, send token}}

[Bandwith(#msg): 2 enter, 1 exit; Client delay(t): 2 msg (req+grant); Sync delay(t): 2 msg (release+grant)]

3. Ring-based Mutual Exclusion [1 token; Safety, eventual liveness; Bandwith: 1@enter, 1@exit; Client delay: best 0, worst N; Sync delay: best 1, worst N-1]

4. Ricart-Agrawala's Algorithm [no token; Liveness worst wait N-1; Casual Ordering; Bandwith: 2(N-1) per enter; Client delay: 1 RTT; Sync delay: 1 msg time] {Enter@Pi: set state "Wanted", broadcast <T (Lamport TS), Pi (break ties)>, wait all other P respond "Reply"; Rcv all "Reply"s @ Pi: set state "Held", enter(); Rcv <Tj, Pj>@Pi: {if "Hold" **OR** "Wanted" & <Ti, i> < <Tj, j>: add req to queue; else: "Reply"}; Exit@Pi: "Released", "Reply" to all queued reqs}

5. Maekawa's Algorithm [Each Pi → voting set Vi (Quorum size K=M=√N), belongs to own Vi AND M other V; Deadlock?; Bandwith: Enter 2√N, exit √N; CD: 1 RTT; SD: 2 MSG] {Enter@P: "Wanted", req to ONLY its V's members, wait to"Hold"; Rcv req@P: {if "Hold" OR voted: queue req; else: "Reply", voted=T}; Exit@P: "Released", multicast "Released" to all Vi members; Rcv Rls@P: {if queue empty: voted=F; else: dequeue head Pk, send Reply ONLY to Pk, voted=T}}

Local Procedure Call [stack pass args & return values; Exactly-once]
Remote Procedure Call [Access procedure via global ref (Proc Addr = <IP, port, proc no.>); ~~pointer~~; **RMI**] [Client: caller(), client sub, comm mod; Server: comm mod, dispatcher(server stub to FW), server stub, callee]

Retransmit request	Filter dup. req.	Re-execute or retransmit	RPC Semantics	
Yes	No	Re-execute function	At least once	Java RMI
Yes	Yes	Retransmit reply	At most once	Sun RPC
No	NA	NA	Maybe	CORBA

Idempotent Operation (can be repeated w/o side-effe)
Middleware **Common Data Representation**; Caller & callee use own platform's way store data; **Marshalling** (caller converts arg into CDR); **Transaction** (series of oper. exec. by client, each a RPC; EITHER completes and Commits all, OR aborts);
ACID [**Atomicity** (all or nothing); **Consistency**; **Isolation** (transaction to be indivisible from point of view of other transactions); **Durability** (永久存储)]

Conflicting Operation (R&W, W&R, R&R)
Serial Equivalence (some ordering O' gives same result as original O; cannot distinguish O' from O) {Two trasactions are SE iff. all pairs of Conf. Oper. are executed in the same order for all objects}

ISOLATION VIOLATION PREVENTION
1. **Pessimistic Concurrency Control** [locking = mutual exclusion] **{**Each object has lock of R/W mode; T first read O: allowed iff all T inside entered via Read; T first write O: allowed iff no T inside; Promote lock R→W; unlock**}** ***Two-phase Locking*** [Growing & Shrinking] **{**T cannot acquire/promote lock after releasing locks.**}**
Deadlocks*: {lock timeout; ~ detection; ~ prevention: {allow read-only; allow preempt.; lock all and abort}}*

2. **Optimistic Concurrency Control** [] {}
First-cut Approach {R/W at will; rollback on abort}
Timestamp Ordering {Each T an ID (pos. in Serialization order); Ensure each T: {W to O allowed if transactions have R/W O had lower IDs; R to O allowed if O was LAST W by a T w/ lower ID}; Abort if violat}
Multi-version Concurrency Control [] {Each O: maintains a per-T and a committed ver.; Each tentative ver. Has a timestamp; Find correct tenta. ver. on R/W}
Eventual Consistency: {@Cassandra & DynamoDB: [**Last-Write-Wins**, unsync clock]; @Riak: [Vector clocks; creates sibling value; size/time-based pruning]}

Storm [Tuple, Stream, Spout, Bolt, Topology {filter, joins, apply/transform}] *Grouping strategy*: {Shuffle ~ (Round-robin); Fields ~ (group by a subset of its fields); All ~}; **Storm Cluster** [Master {***Nimbus***, 分配代码, 派发任务, 监视 failure}, Worker {***Supervisor***, listen for work, run ***Executors***}, Zookeeper {协调 Nimbus 和 Supervisor 通信, 保存它俩的全部 State}]

Graph Processing {每循环: 每节点: {Gather (相邻); Apply; Scatter(更新, send 邻居)}; 循环次数/不再变}
Bulk Synchronous Parallel Model [Hash/Locality]
Pregel System By Google {Master 给每个 worker 分一部分节点, 指导 worker 做一个循环; Halt when no vertix active & no msg in transit; Master 指导 save} [Checkpointing; Failure detection (ping); Recovery]

Replication [fault-tolerance; load balancing; availa-bility] [Transparency (front-end) & Consistency {*Passive Replication* [primary replica]; *Active Replication* [treats all replica identically]}]
Two-phase Commit {Coordinator prepare; Server save to disk, reply "Yes" or "No"; If any "No" OR time out: Abort; Else: Commit, servers "OK" & update to store}

Clustering Coefficient = Pr(A-B | A-C, C-B)

	Extended Ring	Small World	Random
Path len.	大, O(N)	小	小, O(log(N))
CC	大	大	小

Degree distribution (Pr(#edge=k | any node)) {exp @Random: e^(-k.c), power-law@?Small: k^(-a)}

Single Processor Scheduling
1. FIFO/FCFS Sche [Batch app]; 2. **S**hortest **T**ask **F**irst [Optimal!]; 3. **R**ound-**R**obin **S**che. [Interactive; Quick]
Hadoop Scheduling (@YARN)
1. Hadoop Capacity Schedular [有多个 queue, queue 有多个 job; Each queue 分到一部分 cluster capacity] {Soft/Hard limit; Allow preemption (can't stop task part-way); FIFO in queue};

2. Hadoop Fair Scheduler [All jobs equal res.]{Devide cluster into pools (1 pool per user); Res. divided equally among pools; Fair Share Scheduling / FIFO each pool; *Pre-emption* @ min. shares not met: take from others, preempt jobs, killing most-recently-started tasks;)}
Estimate Task Length {running time of input size%;}

Dominant Resource Fairness
[] {Resource vector; 计算 job 的 task 分别消耗不同资源的百分比, 较大者为该 job 的 dominant resource; 使 job i's % d.r. 相等, 解线性方程组;}

File [Header[TimeS, type, owner, ACL, #ref], block…]
Unix File System [file descriptor; open, creat, close, read, write, lseek, link (hard), unlink, stat/fstat]
Distributed File System [Transparency; support concurrent clients; replication] *[Operation be idempotent; Server be stateless]*
NFS {*Virtual File System Module* [allow proc to access file via file descriptors]; Server caching → FAST; delayed write (W to mem., flush to disc) / write-through (W to disc. immediately, consistent but slow); client caching (T-Tvalid<t OR Tmodif_clie = Tmodif_serv)}
AFS [whole file serving & caching (文件小; 读多于写 & sequential)] {client: Venus; server: Vice {sends entire file, gives a callback promise}}

Distributed Shared Memory
Read [O, R/W]:{Read from cache};[R]:{Ask for a copy of page using multicast; Mark as R; Do read;}; [null]: {Ask others to degrade to R using multicast; Get page; Mark as R; Do read;}

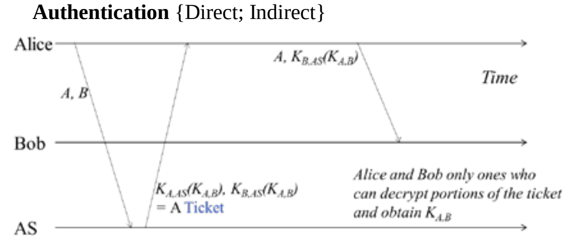
Write [W, O]:{Write to cache}; [R, O?]:{Ask others to invalidate their copies of page using multicast; Mark as W; (Become owner); Do write; }; [null]{Ask others to invalidate their copies of page using multicast; Fetch all copies, use the lastest; Mark as W; Become owner; Do write;}*自己有 page 则其他一定没 W; 自己没 page 则其他可能 R/W.*

Invalidate [better in general] {Write concurrently: flip-flop, (large size) false sharing}; *Update* [lots of sharing; write to small var; page size large] {multicast newly written value / part of page}

TinyOS [event-driven exec.; modular structure] {Better compute than transmit -> In-netwk aggre. (power eff.)}

Security Threats [Leakage; Tampering; Vandalism]
Common Attacks {Eavesdropping; Masquerading; Msg tampering; Replay attack; Denial of service}
Policies [Confidentiality; Integrity; Availability];
Mechanisms {Authenticate; Authorize; Audit}
{Specify Attacker Model, design security mechanism, prove it satisfy policy, measure effect}

CRYPTOGRAPHY SYSTEM
1. Symmetric/Shared Key [DES] {Reveals too much info: hard to revoke perm.}
2. Public-Private Key [RSA; PGP] {Costly en/decrypt}
Authentication {Direct; Indirect}



Digital Signature [SHA-1, MD-5] {[Msg, K_{Apriv}(M)]}
Digital Certificate {$K_{Bankpriv}$(Hash(Name+Account)), $K_{Fedpriv}$(Hash(BankName+PublicKey)), }
Authorization [**Access Control Matrix**, **Access Control List** / obj, **Capacity List** / principal]