# 1 Distributed systems

**Examples**: client/server, the web, the internet, DNS, Gnutella, BitTorrent, "cloud", NCSA, AWS datacenter
**Definitions**: "A collection of independent computers that appear to the users as a single computer", "Several computers doing something together", "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."
**Goals**: heterogeneity, robustness, availability, transparency (hides internal workings from users), concurrency, efficiency, scalability, security, openness

# 2 Clouds

**Types**: public (provide services to any paying customer) or private (accessible only to company employees)
**Benefits**: save time and money!
**Definition**: Lots of storage + compute cycles nearby
**Single-site architecture**: compute nodes grouped into racks; switches connecting racks (both top-of-rack and core); hierarchical network topology
**History**: first datacenters (1940s to 1960s), timesharing and data processing companies (1960s to 1980s), grids (1980s to 2000s), clusters (1980s to present), clouds and datacenters (2000s to present)
**Doubling periods**: storage (12 months), bandwidth (9 months), CPU capacity (18 months)
**Modern cloud features**: massive scale, on-demand access, data-intensive nature, new programming paradigms (MapReduce, Hadoop, NoSQL)
**WUE**: $\frac{\text{Annual water usage}}{\text{IT equipment energy}}$ - low is good
**PUE**: $\frac{\text{Total facility power}}{\text{IT equipment power}}$ - low is good
**"As a service"**: hardware (barebones machines), infrastructure (AWS, Azure), platform (Google App Engine), software (Google Docs)
**Data-intensive computing**: Network/disk I/O more important than CPU utilization
**Academic clouds**: Emulab, PlanetLab
**Public research clouds**: needs grant; Chameleon Cloud, CloudLab

# 3 MapReduce

**Map**: parallelly process lots of records to generate key/value pairs
**Reduce**: processes and merged intermediate values per key; partitions keys for parallelism (shuffle)
**Sorting**: Map output sorted with quicksort, reduce input sorted with mergesort
**Filesystems**: map input (distributed), map output (local), reduce input (multiple remote disks to local), reduce output (distributed)
**YARN**: global Resource Manager (scheduling), per-server Node Manager, per application Application Master (negotiates with RM and NMs, detect failures)
**Locality**: HDFS stores 3 replicas of each chunk (2 on one rack, 1 on another); MApReduce tries to schedule a job on a) a machine with a replica of input, b) on the same rack as a machine with replica of input, or c) anywhere

# 4 Gossip

**Multicast**: disseminates message to group of nodes; nodes can crash and packets can be dropped; protocol should be reliable (atomic, 100% delivery) and fast
**Centralized**: slow ($O(n)$ time to broadcast) and not fault-tolerant (what if sender failed halfway through sending?)
**Tree-based**: build spanning tree; failed nodes may not pass on message until tree is repaired; tree maintenance is hard; ACK/NAK floods could occur
**Gossip**: periodically picks $b$ random nodes and sends messages to them; needs deduplication; all nodes start sending messages as well once "infected"
**Push gossip**: start gossiping after receive message; if multiple messages, gossip random subset, or recently-received, or higher priority; lightweight in large groups, spreads quickly, fault-tolerant
**Pull gossip**: poll randomly selected processes for new messages
**Infection**: continuous time process

$$x_0 = n, y_0 = 1, \frac{dx}{dt} = -\beta xy$$

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

**Epidemic multicast**: $\beta = b/n$, so at time $t = c \log n$,

$$y = (n+1) - \frac{1}{n^{cv-2}}$$

So, within $c \log n$ rounds [low latency], all but $\frac{1}{n^{cb-2}}$ have received the message [reliability] and each node has transmitted no more than $cb \log n$ messages [lightweight]
**Fault tolerance**: Packet loss (50% loss, analyze with $b$ replaced with $b/2$, takes twice as many rounds for same reliabiltiy), node failure (50% fail, analyze with $n/2$ and $b/2$, same as above)
**Pull vs. push**: for both, takes $O(\log N)$ rounds for $N/2$ to get gossip; after first half, pull is faster, as it takes $O(\log \log N)$ time
**Topology-aware**: for random selection, core routers face $O(N)$ load. Fix: pick target in your subnet $i$ with probability $1 - 1/n_i$.

# 5 Failure detection

**Goals**: scalable (equal load per member), fast (time until some process detects failure), complete (each failure detected), accurate (no mistaken detection); impossible to have both accuracy and completeness (or could solve consensus!)
**Preferred**: Guaranteed completeness, partial accuracy guarantee
**Centralized heartbeating**: all ping one node, if no ping from $p_i$ received within timeout, declare $p_i$ failed. Bad: $O(N)$ load on central node
**Ring**: ping neighbors. Unpredictable on multiple simultaneous failures
**All-to-all**: equal load per member, but a single heartbeat loss means a false detection
**Gossip**: nodes periodically gossip membership list to random nodes; on receipt, merged with local membership list; member marked as failed when entry times out, deleted after cleanup period to prevent ghost entries
**Analysis**: heartbeat takes $O(\log N)$ time to propagate, so $N$ heartbeats take $O(\log N)$ time to propagate for $O(N)$ bandwidth per node, $O(N \log N)$ time for $O(1)$ bandwidth, $O(\frac{N}{k} \log N)$ for $O(k)$ bandwidth. If Tgossip is decreased, bandwidth increases, detection time stays unchanged. If Tfail, Tcleanup increased, make fewer mistakes, bandwidth unchanged, detection time increases.
**Load**: All-to-all: $L = N/T$; gossip: $T = \log N \cdot t_g$, $L = N/t_g = N \log N/T$; optimal: independent of $N$!
**SWIM**: every period $T\prime$, $p_i$ sends ping to $p_j$; if $p_j$ ACKs, $p_j$ declared alive; if no ACK, request indirect ping from $k$ random processes
**SWIM detection time**: probability of being pinged in $T\prime = 1 - (1 - \frac{1}{N})^{N-1} = 1 - e^{-1}$, so $E[T] = T\prime \cdot \frac{e}{e-1}$, so complete: any alive member detects failures
**Time-bounded completeness**: round-robin pinging, randomly permute list after each traversal. Failure is detected in worst case $2N - 1$ protocol periods
**Member dissemination**: piggyback on failure detector messages for infection-style dissemination; maintain buffer of recently joined and evicted processes and prefer recent updates
**Suspicion**: per-process incarnation number, only $p_i$ can increment $p_i$'s number; higher #s override lover #s; for given #, suspect > alive, failed > anything else
**In industry**: used by Serf/Consul, Uber's ringpop

# 6 Peer-to-peer systems

**Napster**: napster server maintains <filename, addres, port> tuples but no files; clients upload list of files to server; client queries server for keyword, server responds with list of tuples; clients communicate with each other, selecting the best host for transfer; uses TCP
**Gnutella**: eliminates servers, clients act as servers too; clients connected in overlay graph; messages are routed within the overlay graph
**Gnutella protocol**: *Query* (search), *QueryHit* (response to query), *Ping* (probe network for peers), *Pong* (reply to ping with address of another peer), *Push* (initiates file transfer); header contains unique descriptor ID, type, TTL (decremented at each hop, dropped at 0), hops (incremented at each hop), payload length
**Gnutella search**: flooded out, ttl-restricted, only forward once; *QueryHits* routed on reverse path
**Avoiding excessive traffic**: peer maintains list of recently received messages; query forwarded fo all neighbors except query source; queries forwarded only once; *QueryHit* routed back only to peer from which *Query* received; duplicates are dropped; *QueryHit* with unseen ID is dropped
**QueryHit response**: requester chooses "best" responder, initiates HTTP request directly to peer, peer replies with file packets. HTTP is used (standard, debugged, `Range` header supported
**Firewalls**: if unable to reach peer, routes a *Push* to responder and has the responder send a HTTP-like `GIV` message, requester then sends `GET` as before
**Pingpong**: *Ping* flooded out, *Pong* routed along reverse path; used to update set of neighboring peers periodically
**Problems**: ping/pong constituted 50% of traffic (solution? multiplex, cache, reduce frequency), repeated searches (cache *Query* + *QueryHit* messages), freeloaders (70% of users in 2000), flooding results in excessive traffic
**FastTrack**: like Gnutella, but with supernodes! Peers become supernodes by earning reputation; supernodes store <filename, peer pointer> like Napster; peer searches by querying supernode
**BitTorrent**: files split into blocks; person wanting file first gets tracker from a website, then asks tracker for peers, then gets blocks from peers
**Optimizations**: dowload Local Rarest Block first (early download of blocks that are least replicated among neighbors), tit for tat bandwidth (provide blocks to neighbors with best download rates), choking (limits # neighbors to which concurrent uploads allowed)
**Distributed hash table**: allows lookup, insertion, and deletion of objects (files) with keys
**Perf**: [mem,lookup,#msg for lookup] Napster ($O(1)$ for all, $O(N)$ mem@server), Gnutella ($O(N)$ all), Chord ($O(\log N)$ all)
**Chord**: uses consistent hashing on node addresses; SHA1(address,port) $\leftarrow$ 160 bit string, truncate to $m$ bits; called peer ID, between 0 and $2^m - 1$, maps to $2^m$ logical points on circle
**Finger table**: $i$th entry at peer with ID $n$ is first peer with id $>= (n + 2^i) \mod 2^m$
**Files**: same consistent hash, file with key $K$ stored at first peer with ID $>= K \mod 2^m$; for query, send to largest finger table entry $<= K$, if none exist, send to successor; repeat, takes $O(\log N)$ time total
**New peers**: $O(\log N)$ finger entries affected, $O(\log N * \log N)$ messages per join (similar for leave); stabilization protocol that checks and updates pointers and keys to ensure non-loopiness; strong stability takes $O(N^2)$ stabilization rounds
**Pastry**: virtual ring with node IDs, like Chord; routing tables based on prefix matching (think: hypercube), so routing is $O(\log N)$ and network hops are short
**Routing**: nodes maintain list of neighbors matching prefixes of its own address with the $n$th bit differing; to route to a peer, forward to neighbor with largest matching prefix
**Kelips**: 1-hop DHT; $k$ affinity groups, $k \approx \sqrt{N}$; nodes hashed to groups $\mod k$, each group has one contact node in each other group

# 7 Key/value stores & NoSQL

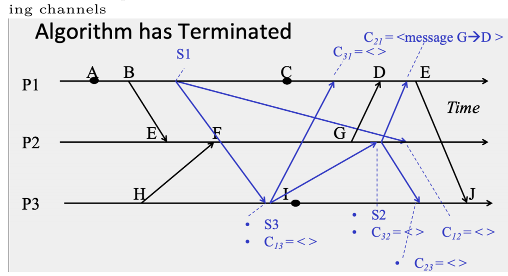
**Abstraction**: basically a dictionary (insert, lookup, delete by key)
**Relational databases**: don't match today's workloads (large & unstructured data, lots of random reads/writes, foreign keys rarely needed, joins infrequent)
**Modern workloads**: need speed; avoid single point of failure; scale out, not up
**NoSQL**: "Not Only SQL"; API needs get(key) and put(key, value); sometimes extended SQL-like operations (see "CQL" for Cassandra)
**Tables**: "column families" in Cassandra, "tables" in HBase, "collections" in MongoDB; often unstructured (no schema); don't necessarily support joins or foreign keys, but can have indices like RDBMs
**Column-oriented storage**: NoSQL typically store column or group of columns together, columns are indexed for fast lookup; useful because range searches can be fast (no need to fetch whole DB)
**Cassandra**: Ring-based DHT, but no finger tables or routing; offers eventual consistency (replicas converge if writes stop)
**Placement strategies**: simple (Random partitioner, like Chord; ByteOrdered, assigns ranges of keys to servers); network topology (multi-DC deployments: first replica per Partitioner, then go clockwise until different rack)
**Snitches**: Map IPs to racks and DCs; simple (rack-unaware), rack inferring (x.DC octet.Rack octet.Node octet), property file (config), EC2 (region = DC, availability zone = rack)
**Writes**: send write to one coordinator in cluster, coordinator talks to replicas; once $X$ replicas respond, ack to client; if any replica is down, buffer until back online, then write (buffer up to a few hours); one ring per DC
**Writes at replicas**: log to disk; change memtable (in-memory representation of multiple K/V pairs, write-back); flush to disk as SSTable (immutable, sorted by key, uses bloom filter) when full
**Bloom filter**: set all hashed bits, to query, check if all hashed bits are set
**Compaction**: merges SSTables by key
**Deletes**: add tombstone; compaction will delete later
**Reads**: coord contacts $X$ replicas, when $X$ respond, return latest value
**CAP**: any two of consistency, availability, partition-tolerance
**BASE**: Basically-Available Soft-state Eventual consistency
**Consistency levels**: ANY, ALL, ONE, QUORUM
**Quorum**: majority (> 50%), any two quorums intersect; $W + R > N$, $W > N/2$
**W,R**: W=R=1 for few R/W; W=N,R=1 for R-heavy; W=R=N/2+1 for W-heavy; W=1,R=N for W-heavy with one client/key
**Linearizability**: each operation available simultaneously to all clients
**Sequential consistency**: result of any execution is same as if operations of all processors were executed in some sequential order, and operations of each processor appear in sequence in the order specified by program (find reasonable order after the fact)
**Per-key sequential**: per key, all operations have a global order
**CRDTs**: Cumulative Replicated Data Types; commutated writes give same results
**Red-blue consistency**: blue ops executed in any order across DCs, red ops executed in same order at each DC
**Levels**: Eventual, casual, red-blue/per-key, probabilistic/CRDTs, strong/sequential
**HBase**: Get/Put, Scan(range/filter), MultiPut; table split into regions (replicated); ColumnFamily (subset of columns with similar patterns); one Store per ColumnFamily+region; Memstore per store (flushed to disk when full); StoreFiles for each region; HFile (list SSTable); strong consistency via write-ahead log; replay stale logs after failure recovery or bootup (select via timestamps); coordination via Zookeeper
**MongoDB**: BSON (Binary JavaScript Object Notation) documents; group of related docs with common index called a collection; data split in chunks, sharded into collections of chunks, shards assigned to replica set, which is multiple servers (usually 3), replica set members mirror each other (one is primary, other secondaries); routers send queries to correct replica set; oplog to sync data; read concern (primary, primary-preferred, secondary, nearest, majority) - secondary might fetch stale data; write concern (0 no ack, 1 primary ack, majority) - weaker concern implies faster write; journal in write-ahead log for durability; chunks split when grow too large, and migrated among shards if uneven distribution; either strongly or eventually consistent, depending on R/W concerns

# 8 Time and ordering

**Clock skew**: relative difference in values
**Clock drift**: relative difference in rates/frequencies
**Synchronization**: max drift rate (MDR); max drift between two clocks with same MDR is 2MDR; for acceptable skew $M$, clocks must sync ever $M/(s * \text{MDR})$ units
**Cristian's**: $P \to S = \text{min1}$, $S \to P = \text{min2}$; $T = [t + \text{min2}, t + \text{RTT} - \text{min1}]$, err $< (\text{RTT} - \text{min2} - \text{min1})/2$
**NTP**: offset $= (\text{tr1} - \text{tr2} + \text{ts2} - \text{ts1})/2$
**Lamport**: happens-before $= \to$; rules: a) $a \to b$ in same process is time$(a) <$ time$(b)$, b) send$(m) \to$ receive$(m)$, c) transitivity; creates partial order
**Impl**: process keeps local counter; increments on send or instruction; messages carry timestamps; on receive, counter becomes max(local, msg timestamp) + 1
**Concurrent**: no causal path from one event to the other; Lamport cannot distinguish concurrent events
**Vector timestamps**: process $i$ maintains vector $V_i[1 \ldots N]$; when $i$ has instruction or send, increment element $i$; messages carry entire vector; when receive at $i$, $V_i[i]+ = 1$, $V_i[j] = \max(V_{\text{message}}[j], V_i[j]) \forall j \neq i$
**Causally-related**: iff $VT_1 < VT_2$, i.e. $VT_1 \leq VT_2$, and $\exists j$ such that $1 \leq j \leq N$ and $VT_1[j] < VT_2[j]$; events are concurrent iff NOT $VT_1 \leq VT_2$ AND NOT $VT_2 \leq VT_1$

# 9 Snapshots

**Def**: individual state of each process and communication channel; assume FIFO and no failure or message duplication
**Chandy-Lamport**: initiator $P_i$ records state; send Marker message on all outgoing channels and start recording incoming messages on each incoming channel; when msg received, a) first Marker, so record state, mark incoming channel empty, send marker message to other processes, record on incoming channels except self and source of marker; b) already seen, so mark state of incoming channel as all arrived messages since recording started; ends when all processes have received marker on all $N - 1$ incoming channels




**Cut**: time frontier at each process/channel; events before cut are "in the cut"; cut is consistent if it obeys causality: for all pairs $e, f$, if $e$ in cut and $f \to e$, then $f$ also in cut; Chandy-Lamport snapshot creates consistent cut

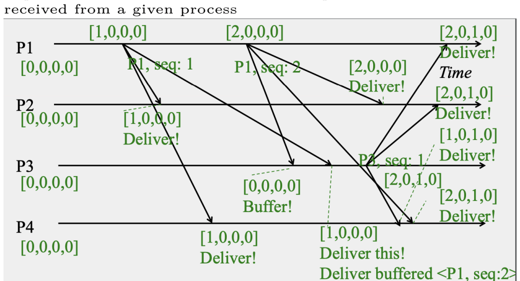
# 10 Paxos

**Goals**: safety (no disagreement), liveness (protocol ends)
**Synchronous**: with at most $f$ crashing, run $f + 1$ rounds; $\text{Values}_i^r$: proposed values known to $p_i$ at beginning or round $r$; $\text{Values}_i^0 = $ ; $\text{Values}_i^1 = v_i$; for rounds 1 to $f + 1$, multicast$(\text{Values}_i^r - \text{Values}_i^{r-1}$, $\text{Values}_i^{r+1} \leftarrow \text{Values}_i^r$, for each $V_j$ received, $\text{Values}_i^{r+1} = \text{Values}_i^{r+1} \cup V_j$; $d_i = \min(\text{Values}_i^{f+2})$
**Paxos**: provides safety, eventual liveness; asynchronous rounds; 3 phases: election, bill, law
**Impl**: election: potential leader chooses unique higher ballot ID, send to all processes, processes respond once to highest ballot ID, if process has previously decided $v\prime$, include $v\prime$ in response, become leader when a quorum responds with OK; bill: leader

sends proposed $v$ to all, use $v = v\prime$ if received during election, recipient responds OK; law: if leader hears majority OKs, tell everyone decision. decision reached when majority of processes are about to/have responded with OK
**Handling problems**: P restarts (use log to retrieve a past decision and past- seen ballot ids), leader fails (start new round), anyone can start a round any time, may never end

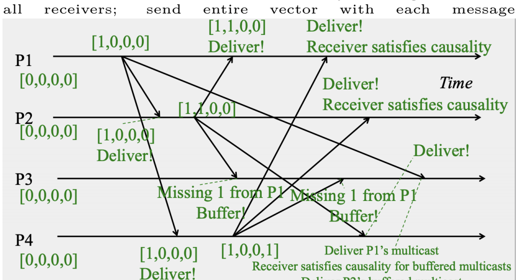
## 11 Multicast

**Forms**: multicast (to group), broadcast (all processes), unicast (one-to-one)
**FIFO ordering**: multicasts from each sender received in order they are sent, at all receivers; each receiver maintains vector of sequence numbers that is the latest sequence number that was received from a given process




FIFO Ordering: Example

**Causal ordering**: send events that are causally-related must be received in same causality-ordering order at all receivers; send entire vector with each message




Causal Ordering: Example

**Total ordering**: aka "atomic broadcast", ensures all receivers receive all multicasts in the same order (if $P$ delivers message $m$ before $m\prime$, then any $P\prime$ that delivers $m\prime$ would already have delivered $m$); sequencer approach: $P_i$ sends $M$ to group and sequencer, sequencer maintains $S$ and increments/broadcasts it when receives a message; $P_j$ buffers until sequencer message received and local sequence counter + 1 = received sequence number
**Hybrid**: FIFO/Causal orthogonal to Total, hybrid approaches that satisfy both can exist
**Virtual/View synchrony**: combines membership+multicast protocol
**Views**: each process maintains a membership list (view); VS guarantees all view changes are delivered in same order at all correct processes; VS ensures a) set of multicasts delivered in a given view is the same set at all correct processes that were in the view (what happens in view stays in view), b) sender of multicast message also belongs to view, and c) if $P_i$ does not deliver $M$ in $V$, while other processes in $V$ delivered $M$ in $V$, then $P_i$ will be forcibly removed from next view at other processes
**Problems**: susceptible to partitioning

## 12 Leader election

**Goal**: elect exactly one agreed-upon leader
**Guarantees**: safety (all non-faulty processes choose same process or null), liveness (all election runs terminate and elected process is not null)
**Ring election**: process discovering failed coordinator sends "election" msg with own ID; for received msg: a) ID is greater, so forward, b) ID is smaller and $p$ not forwarded an earlier election message, overwrite with own ID and forward; c) if ID is same as process, $p$ is now leader, sends "elected" msg; safe and live if no failures; best case $2N$ messages, worst case $3N - 1$ messages
**Bully**: all processes know all IDs; process discovering failed coordinator does a) if knows its own ID is highest, sends "coordinator" message to all processes with lower IDs, or b) sends "election" message to processes with higher IDs. If no answer within timeout, becomes leader and sends "coordinator" to all lower IDs. If answer, wait for "coordinator" message; if none after timeout, start new election. Process receiving "election" replies "OK" and starts own leader election protocol (unless already done so); safe if failures stop; worst case $O(N^2)$, best case $(N - 2)$ "coordinator" messages
**Consensus to solve election**: each process proposes a value, group reaches consensus on $P_i$'s value, $P_i$ becomes leader
**Chubby**: potential leader asks for votes from other servers, each server votes for at most one leader, server with majority (quorum) becomes leader, informs others

## 13 Mutual exclusion

**Uses**: DFS; safe/consistent access to objects; server coordination
**Properties**: safety (essential, only one process in CS at any time), liveness (essential, every request granted eventually), ordering (desirable, requests granted in the order received)
**Semaphore**: allows max accessors, wait/signal
**System model**: reliable channels, FIFO, no failure
**Perf**: bandwidth (total messages sent per enter/exit operation), client delay (incurred at each enter exit operation, assuming no process in or waiting), synchronization delay (interval between one CS exit and next CS enter, assuming only one waiting)
**Central**: master keeps queue of waiting requests and a token; sends token to process on its turn; process sends token back on CS exit; safe, live, and FIFO!; bandwidth (2 enter, 1 exit), client (2 msg latencies for req + grant), sync (2 msg latencies); but master is bottleneck/SPoF
**Ring-based**: 1 token, wait til get token, pass token on exit, pass on if not waiting; safe and live; bandwidth (up to N in system on enter, 1 on exit), client (best case, have token; worst case, just sent token to neighbor), sync (1 to $N - 1$)
**Ricart-Agrawala**: multicast request with Lamport timestamp, wait for all to respond OK; requests granted in order of causality; $2(N - 1)$ messages per enter, client delay = 1 RTT, sync delay = 1 message transmission time
**Maekawa**: process associated with voting set; intersection of any two voting sets must be non-empty (quorums!), $K$ sets, $M$ sets/process, $K = M = \sqrt{N}$ is best; $2\sqrt{N}$ per enter, $\sqrt{N}$ per exit, client delay = 1 RTT, sync delay = 2 message transmission times
**Chubby**: Paxos-like consensus; advisory locks only (no mutex guaranteed unless every client checks locks before access)

## 14 RPCs & concurrency control

**Local Procedure Call**: exactly-once semantics
**RPC Semantics**: at most once (Java RMI), at least once (Sun RPC), maybe/best-effort (COBRA)
**Idempotent**: applied multiple times without any side effects; can be used with at-least-once semantics
**Marshalling**: convert req/res into common, platform independent representation
**Transactions**: either completes and commits all operations, of aborts with no effect
**ACID**: atomic (all or nothing), consistency (if starts in consistent state, transaction ends in consistent state), isolation (transactions performed without interference from other transactions), durability (all effects saved in permanent storage after transaction completed successfully)
**Serial equivalence**: interleaving of transactions is serially equivalent iff some ordering of transactions which gives the same end result as the original interleaving
**Conflicting operations**: if combined effect depends on order of execution (R then W, W then R, W then W)
**Checking serial equivalence**: iff all pairs of conflicting operations are executed in the same order for all objects they both access (label all pairs $(T1, T2)$ or $(T2, T1)$, all pairs should be marked the same)
**Upshot**: at commit point, check for serial equivalence with all other transactions' if not equivalent, abort $T$ and roll back writes that $T$ did
**Pessimistic CC**: prevent transactions from accessing same object (locking; use R/W locks for better perf)
**Optimistic CC**: assume the best, allow transactions to write, but check later; higher concurrency and transactions per second
**Two-phase locking**: transaction cannot acquire or promote any locks after it has started releasing locks (growing/shrinking phase; strict: only release locks at commit point); proof of serial equivalence by contradiction (growing/shrinking phases would overlap)
**Deadlocks**: mutual exclusion, preemption, circular wait all necessary
**Basic optimistic CC**: check serial equivalence at commit time, roll back if abort (problem: cascading aborts)
**Timestamp ordering**: assign each transaction and ID; ID determines position in serialization order; ensure for $T$ a) $T$'s write to $O$ allowed only if transactions that have R/W $O$ had lower IDs than $T$, and b) $T$'s read to $O$ allowed only if $O$ was last written by transaction with lower ID than $T$; abort if violation
**Multi-version CC**: maintain per-transaction tentative version and committed version; each tentative version has timestamp; on R/W, find "correct" tentative version to R/W from (tries to make transactions only R/W from "immediately previous" transactions
**Eventual consistency**: form of optimistic concurrency control in K/V stores
**Cassandra/DynamoDB**: last write wins (overwrite only if new write's timestamp > current timestamp)
**Riak**: vector clocks! Implements causal ordering, detect if a) new write is strictly newer than current value, or b) if new write conflicts with existing value. For b) *sibling value* is created to be resolved by user/application; size/time based pruning to prevent clocks from getting too many entries

## 15 Replication control

**Motivation**: fault-tolerance, load balancing, availability (with replication, probability of available replica is $(1 - f^k)$), transparency (client doesn't know multiple copies exist), consistency (all clients see consistent copy of data)
**Passive replication**: use primary replica
**Active replication**: treat all replicas identically; multicast inside replica group with any type of multicast ordering; handle failures with virtual synchrony
**One-copy serializability**: true if equivalent to a serial execution of transactions over a single logical copy of the database
**Transactions**: might touch object on multiple servers, need atomic commit (consensus!)
**One-phase commit**: coordinator server initiates atomic commit
**Two-phase commit**: coordinator sends "prepare"; servers save updates to disk, respond with Y/N; if coordinator receives all Y within timeout, tell servers to commit updates from disk to store; otherwise, abort all
**Paxos**: server proposes message for next sequence number, group reaches consensus (or not); allows for "no" votes

## 16 Stream/graph processing

**Motivation**: have lots of data, want high throughput and low latencies
**Storm**: tuples, streams, bolts, spouts, topologies (cycles possible)
**Bolts**: parallelize by splitting streams among multiple tasks (grouping: shuffle, fields, all)
**Cluster**: master (runs Nimbus, distributes codes, assigns tasks, monitors failures), worker (runs supervisor, receives work, runs executors that run groups of tasks), Zookeeper (coordinates communication, stores state)
**Twitter Heron**: uses backpressure, better throughput
**Spark**: Resilient Distributed Datasets (RDDs), immutable partitioned collections of records built through map/join/etc.; log operations and recompute lost portions if failure occurs
**Graph processing**: shortest paths (routing, degrees of separation), matching (dating), PageRank
**Algorithm**: work in iterations; each vertex gets a value; in each iteration, a vertex a) gathers values from immediate neighbors, b) does computation with own/neighbor values, c) updates value and sends to neighboring vertices (gather, apply, scatter); terminates after fixed iterations or values stop changing
**Processing**: assign each vertex to a server; perform gather-apply-scatter for all assigned vertices; assign based on hash or locality (assign vertices with more neighbors to same server as neighbors, reduces traffic after each iteration)
**Pregel**: master/worker (P2P workers); data persisted on DFS; fault-tolerance via checkpointing and recovery; fast!

## 17 Scheduling

**Goals**: throughput, high utilization of resources
**FIFO/FCFS**: queue in arrival order, execute when processor free (average completion time might be high)
**STF**: order tasks by running time, run shortest task first (optimal (minimal average completion time), can lead to starvation; special case of priority scheduling)
**Round-robin**: run portion of task, preempt after quantum expires, add to end of queue (preferable for interactive applications)
**Hadoop Capacity Scheduler**: multiple queues (possibly hierarchical), each queue contains multiple jobs; each queue guaranteed some portion of cluster capacity; elasticity (can occupy more resources if they're free; no preemption
**Hadoop Fair Scheduler**: goal is for all jobs to get equal share of resources; cluster divided into pools, resources divided equally between pools; configurable scheduler within pool; preemption allowed (kill most recently started tasks to minimize wasted work; OK because tasks are idempotent)
**Estimating task lengths**: hard; can estimate (proportional to size of input; weighted average by input size across other tasks in job)
**Dominant-Resource Fair Scheduling**: tries to handle multi-resource requirements (mem/CPU/etc.); jobs have resource vectors $< N$ CPUs, $M$ GB RAM $>$; for a given job, the % of its dominant resource type that it gets cluster-wide is same for all jobs (Job 1 RAM dominant, Job 2 CPU dominant, Job 1 % RAM = Job 2 % CPU)

## 18 Structure of networks

**Properties**: Clustering coefficient ($P(AB|(AC\&\&CB))$, path length (shortest path); Extended right graph (high CC, long paths), Random (low CC, short paths), Small world (high CC, short paths)
**Small-world resilience**: most nodes have small degree, few nodes have high degree; killing lots of random nodes won't disconnect, but a few high-degree nodes will

## 19 Distributed file systems

**Unix FS**: file descriptors (handle for process to access FS); must open file before reading/writing; descriptor maintains R/W pointer to offset within file; R/W automatically advances pointer
**Distributed FS**: client does RPC to perform file ops; desirable properties: transparency (behaves like local FS), concurrent clients, replication (fault tolerance)
**One-copy update semantics**: when file is replicated, contents visible to clients are no different from when only 1 replica
**Client API**: read/write specifies absolute position and num bytes, position maintained by client (should be idempotent and stateless, unlike Unix)
**NFS**: client integrated with kernel, performs RPC to server; server allows mounting files and directories (essentially, pointers); allows processes to access file descriptors (transparency!); server caching (store recently access blocks in memory (supports locality of access)); writes: delayed (flush every 30s, fast but not consistent), write-through (write to disk immediately, consistent but maybe slow); client caching ($Tc$ = time of cache entry last validated, $Tm$ = time when block last modified at server, block valid at $T$ if $(T - T_c < t)$ or $(Tm_{\text{client}} = Tm_{\text{server}})$, delayed write to server on write)
**AFS**: whole file serving/caching (based on assumptions that most files accessed by single server, most files small, files read more often than written, reads typically sequential), Venus client, Vice server, optimistic reads/writes, opened files get callback promise (Vice notifies Venus if another client modifies/closes)

## 20 Distributed shared memory

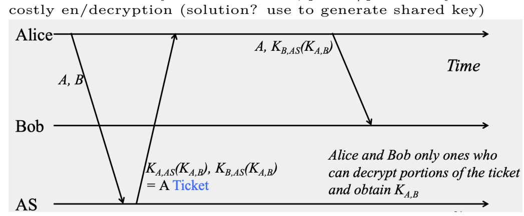
**Benefit**: same code as if running on same multiprocessor OS
**Impl**: cache maintained at each process, stores recently accessed pages; pages mapped in local memory; on page fault, kernel invokes DSM
**Protocol**: Owner is process with latest version; page in R or W state; when page in R state, owner has R copy, others may have R copies, no W copies; when page in W state, only owner has copy
**Read**: [O?, R]: read from cache; [null, other R]: ask for copy, mark R, do read; [null, other W]: ask other to degrade to R, ask for copy, mark R, do read
**Write**: [O, W]: write to cache; [O, R]: invalidate others, mark W, do write; [~~O~~, R]: invalidate others, mark W, become owner, do write; [null]: invalidate others, fetch latest copy, mark W, become owner, do write
**Invalidate downsides**: two processes writing concurrently (flip-flopping, *false sharing*)
**Update**: multiple can have in W state; on write, multicast updated part of page; preferred when lots of sharing, writes are to small variables, or large page sizes

## 21 Sensor networks

**Node**: sensors, microprocessor, comms link, power
**Transmission**: RF (store and forward, bidirectional), optical (directional antennas, costly broadcast, line of sight needed, passive routing ("wormhole"), unidirectional links)
**TinyOS**: event-driven (hardware events), modular structure (components), static allocation
**Power saving**: MICA (active, idle, sleep)
**Optimization**: transmit is expensive, so compute instead of transmit; build trees among nodes, calculate summaries in trees, transmit only summaries

## 22 Security

**CIA**: confidentiality (no unauthorized disclosure), integrity (no unauthorized alteration), availability (data always readable/writable)
**Terminology**: policy (what a secure system does), mechanism (how the system accomplishes goals)
**Golden A's**: authentication, authorization, auditing
**Symmetric key systems**: $K_{AB}$ shared only by Alice and Bob (Data Encryption Standard)
**Public-private key systems**: $K_{A\text{priv}}$ known only to Alice, $K_{A\text{pub}}$ known to everyone (RSA, PGP)
**Details**: Shared keys hard to revoke; public/private keys involve costly en/decryption (solution? use to generate shared key)




**Signatures**: authentic, unforgeable, verifiable, non-repudiable: $[M, K_{A\text{priv}}(\text{Hash}(M))]$ (hash for efficiency)
**Authorization**: Access Control Matrix (large & sparse), Access Control Lists (per object, allowed principals/access), Capability Lists (per principal, list of files & access types)