Time uses - scheduling jobs cos, Hadoop) - timeout & retry - failure detection - measuring performance - TTL caches, time-sensitive data - Order of events-timestamp weak to internet down

Clock Types - physical (analog = pendulum, digital = oscillating crystal) - crystal cheap, inaccurate, temp-dependent - atomic uses cesium-133 or rubidium, GPS - adjusted to match Earth's variations (GTC) - June 30 & Dec 31 ± leap sec every year (23:59:58 → 00:00:00 or 23:59:59 → 60 or nop) - OS livelock @ 23:59: 60 confusion, disrupts hr timer to think all sleeping processes were sleeping too long due to OS neglect, wakes them all, overloads CPU with callbacks in Java - CMOS battery counts time even if PC off, crashes badly if it fails - fixing clock skew: minor inaccuracies btwn 2 computers - **NTP servers** strotums: - 0: atomic clock - 1: Ntp server built atop 0 - 2: everything else built atop 1 - ==**clock slew**==: gradually adjust client clock - gradually adjust bc sudden step may miss milestones or confuse hr timer to livelock![[Pasted image 20241102172029.png]] - if <125ms slew, elif <1000ms step bc too slow, else too big ask for human intervention - assumes symmetric, may not due to queue, so average out many requests? - stepping clock skew issues due to diff types of software clocks - ==time-of-day (real-time)==: i.e. 02:00:00 PDT, error-prone, leap-second minus/plus etc - can be compared across nodes if in sync UNLIKE monotonic clock (but problem of clock skew) - Cloudfaire DNS 2017 huge outage bc not use monotonic clocks and stepped backward = negative duration - ==monotonic clock==: only counts forward, # seconds since arbitrary event (i.e. since awoke)–> nanoTime() in Java, - less prone to drift (hardware error that cause clock skew) - good for timing local stuff, but arbitrary means in relation to this computer only - ordering messages - uses? for debugging. serialization - if b has unexpected X=6, find all a where a–>b and see if a influenced/caused X=6 (i.e. ++) **rule-out** causes - bank example of issue: - bank's data balance ($1000) replicated across two nodes, LA & NYC - LA user wants to deposit $100, NYC employee deposits 1% interest - LA then NYC: $1000+$100+$11=$1111 - NYC then LA: $1000+$10+$100=$1110 - how to find order if jumbled due to latency? - monotonic relative can't, real-time timestamps from each user non-synced/skewed - total order: no two events same timestamp/seqn or no dupes, vs partial order concurrency - soln: ==**Happens-Before**== (type of partial ordering) - a –> b (a before b) iff: - single thread/node/process responsible for both a and b and a executed first ![[Pasted image 20241102174858.png]] - or a = "sending msg" and b = "receiving msg" ![[Pasted image 20241102174922.png]] - or exists event c such that a–> c and c–>b. i.e. given image: - a–>c because a–>b and b–>c - a–>d because a–>b and b–>c and c–>d - etc. - exercise: ![[Pasted image 20241102192039.png]] - X –> Z? yes because both by B - R –> Z? no, R||Z because R is dead end / no path from R to Z - otherwise a||b, aka **concurrent**/independent, not know which first or not) - a||e, b||e, c||e, d||e. because all we know is e–>f but nothing is before e - broadcast: lots of complexity and order matters! - logical clocks - designed to capture causal dependencies - # events NOT seconds - why? so processes understand # events b4 and causality/concurrency (DAG) - built in, instead of central coordinating node - ==Lamport clocks== - seq no. not timestamp, logically monotonic - a–>b => L(A) < L(B) - why useful: if negate property both sides, use seqno to rule out a->b but can't rule out b->a or a||b - distrib databases, event order, versionings in GFS, etc. but mostly outdated by vector clocks or its improvements. - assumes: single thread per node - each node counter t per local event, so O(E) space and time - cons: not very scalable, not fault-tolerant if one node dies - L(e) = current val of t at event e (right after incrementing to e) 1. t=0 2. before executing/send/deliver event, increment that node's t 3. when node i –> j a message, t_i included in msg, then node j updates t_j = max(t_i, t_j)+1 - if a –> b, then L(a) < L(b) but not guaranteed vice versa: - ![[Pasted image 20241102193152.png]] - here A's 1 < B's 5, but path up DAG shows 5 not dependent on A's 1. - psuedocode: - ![[Pasted image 20241102180554.png]] - example: ![[Pasted image 20241102182201.png]] - properties: a –> a - what about conflicts with A's 1 and C's 1? independent - sort by alphabet (1,A), (1,C), (2,A), (3,A), (3,B), (4,B), (5,C) - still not happens-before execution ordering, just algorithm output - even if L(a) < L(b) still cannot conclude causal order: - ![[Pasted image 20241102183411.png]] - ==Vector Clock== identifies a||b from a–>b using seq no - a –> b <=> V(a) < V(b) - usages: DynamoDB for transactions, concurrent writes, Git, message systems in queue, google docs conflict edits, etc. - difference is it tracks each other's numbers too - element-wise maximum between msg vector and own vector whenever receives message - ![[Pasted image 20241102183941.png]] - scales badly in RAM and I/O (sending vectors) - a–>b iff - all counters in T1 <= their corresponding counters in T2 - one or more counters in T1 < corresponding counter in T2 - ![[Pasted image 20241102193833.png]] - red = caused A.

green = caused by A. notice all reds’ elements are <= A’s and >= for green. - a||b iff - exists T1[i] > T2[i] and T2[j] > T1[j] incongruent/out of step from each other. - aka A’s (1,0,0) and C’s (0,0,1) hence inconsistent as i=0 and j=2![[Pasted image 20241102183941.png]] - properties - if A and B have same vectors, A and B are same node -