Lamport Clocks:

First, questions about project 1: due date for the design document is Thursday. Can be less than a page – what we're after is for you to tell us what you are planning to do, so that we can take a look and say, "hm, sounds good" or "uh, maybe you want to rethink that a bit." There's a bit of skill on your part needed: how do you describe something you plan to do, succinctly, so that you can get useful feedback?

Lamport: Can we make sure everyone agrees on the same order of events?

Why does this matter? In a client-server system with one server, not so important. We don't really care if one client or the other sent their RPC a bit before the other one.

But as we'll see that's not the common case. E.g., what does Facebook's storage system look like? Initially, a few clients and a server. Then more clients, more servers. Each client draws from information across the various servers; are they seeing a consistent view? E.g., if I update my privacy settings and then do a post, does everyone either view the post according to the new privacy settings? How can I guarantee that if I have no definition of global time?

Now, add some caches! FB says they do approx 100 separate lookups per page; can't possibly go to the server. And they have approx 1M servers, and another 1M caches.

This is where Lamport fits in – to help explain what you can say about the order of events in a distributed system.

We'll start with a really simple case: no failures! Messages can only have bounded delay, and all we're interested in, is making sure multiple hosts come to some common understanding about their shared state. This should be easy, right?

Example #1: going out to lunch. Order of operations changes the semantics, so we only have shared knowledge, if we see the same ordering of events.

Arranging lunch – value of preserving the causal order.

a: how about lunch? meet at 12?

b: can't – meeting. 1pm?

c: <receives a>: ok by me

d: <receives c>: what's ok?

a: <receives b then c>: which is ok?

Solution: if messages are delivered in same order everywhere, no opportunity for confusion.

Example #2: a shared whiteboard (CSCW). Lets assume processor order – in the paper, what's meant by events at one processor are delivered in that processor's order (no message bypassing). Problem is if two of us are listening to two people drawing in the same part of the WB at the same time. If I receive erase, then draw, and you see draw, then erase, it will appear quite different. And users, since they only see their own view, won't be able to figure out the problem. How might we fix this?

Example #3: Parallel make. Distributed set of computers, updating a set of files, such that system rebuilds any derived object if it is out of date. How can it tell? Time stamp on each file? But time stamps aren't accurate, unless system makes them so.

Example #4: iPhone scheduling app, that works even when disconnected. What if multiple iPhones come close to each other, so they can synch their calendars, but not with others in group? How do you resolve the potential confusion?

You might think that you can just assign every event in the distributed system a real-time stamp.

But that doesn't work, since clocks aren't perfect – they need to be synchronized, they drift with respect to each other, and the only thing we have to do the re-synchs, are unreliable messages of uncertain performance.

Lets assume for simplicity that there are some nodes in the system have well-defined wall-clock time, that is, they have a GPS unit.

Even with a GPS, you need to distribute the GPS information over the network to the other nodes in the system. How close can we bound the clocks on those other systems?

Option 1: if there's an upper bound on the message time, can synch to 2max RTT

but there (often) isn't an upper bound in practice (or its absurdly high). Even on a LAN, RTT > 1M instructions

Option 2: keep fetching the time, estimate relative rate of skew of the local clock. Can give a somewhat tighter bound.

Do either of these solve the problem with the makefile timestamps?

<We'll see later on with Google's Spanner a system that takes the approach -- events are only located within a window of time of several milliseconds, and the system needs to be able to handle that.>

If no external events, don't need physical time, so Lamport argued for "logical clocks".

Very simple version of logical clocks: complete centralization: send every message to a central arbiter, which picks an order for all messages, and then system can deliver them in that order.

Problems with centralization? Obviously scale – it wouldn't work for the FB example we started with. But also, need to somehow recover if/when the central arbiter fails.

Is a central arbiter required for consistent delivery everywhere? Lamport proposes an alternate algorithm.

Introduce "space time diagram": 3 processes, each does an RPC

p1, p2, p3: p2->p1, p1->p3, p3->p1, p3->p2, p1->p2, p1->p3

p2 does RPC to p1; p1 does RPC to p3; p3 replies to p1; p3 does RPC to p2; p1 replies to p2; p1 does RPC to p3

Example #5: This can appear to be a deadlock! At various points, p2 waits for p1; p1 waits for p3; p3 waits for p2. How can we tell what is live? (Need a consistent snapshot.)

A bit of notation: happened before: a->b [b could be caused by a]

1) if a and b are in same process, a before b
2) send before receive
3) transitivity

We can use this to build a logical clock, C. Cj is the value of clock C for process j

such that if "a happened before b" then C(a) < C(b)
[note I haven't defined < ] sidebar on PL operator redefinition, aka C++ can refine [

IR1: increment on every event within a process
IR2: put current timestamp in each message; on receipt, Cj = max(current, message + 1)

What is the logical clock for the example above? (assign timestamps to each event.) Counter just keeps going up; ok if multiple (concurrent) processes have a different idea of the current time.

If we define a tiebreaking rule, then we get a total order consistent with partial order.

We can then use this to build workable distributed algorithms.

Can we use something like this to solve the whiteboard example?

Always deliver packets in the same total order (defined by Lamport clock). Will this work? Even if all nodes are awake and working, need to wait to display until ping neighbors, since you might get a message from B that depends on A, but you haven't seen A yet.

Painful, since most of the time, no one else is editing the same part of the whiteboard as you are.

GUI design principle: always allow user to edit (AJAX). How do we do that while preserving correctness (whiteboards do not stay out of synch)?

Keep timestamped log of all edits to whiteboard. Can apply changes in timestamp order. If an "old" update comes in, insert in log and redraw.

Paper uses example of mutual exclusion.
Correctness:
one at a time
everyone in order of request
Liveness: (in absence of failures: if every holder releases it) eventually get the lock

Implementation: Every message timestamped with value of logical clock.

1) multicast request Tm to everyone
2) if get request, put Tm on local queue, ack (so now request is on everyone's queue)
3) to release, remove request from local queue, send release to everyone
4) if get release, remove that request from local queue
5) Process i gets lock if its request is before (=>) everyone else's, and none earlier can arrive (have a later timestamp from everyone)

Example of state machine; everyone operates on the same information, so everyone stays in synch.

Example: start with set of processes, each with a request queue.

P0 starts with lock, at T0. In everyone's request queue.
P1 asks for lock, at T1.
P2 asks for lock, at T2.
P0 releases lock, at T2.

Any problems? What if there are failures?

We'll see how to generalize this to handle failures with paxos.

Suppose I only want to avoid causality anomalies, but I don't need a total order. That's all that's needed for the lunch example – that no message that depends on a past event is delivered before that event.

Might need to ping neighbors to see if their timestamp has exceeded yours. <Problem? If someone has already left for lunch, then no one else gets to go to lunch!>

Another implementation: vector clocks.

Vector clock: what is latest event I have seen from each process? VC[i] is the latest event at process i

Pass vector clock in each message, and update local clock on message receipt (same as before, only element by element max).

For all i, VC[i] = max (VC[i], msg-VC[i] + 1)

VC defines a partial order for events, but can be used to enforce constraint that you don't process an event until you've seen all prior events from each process.

This helps with constructing consistent snapshots, a topic I'll discuss next time. Snapshot is consistent if taken at the frontier of VC[i] – no events that depend on any that are missing.