Lamport Clocks continued:

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

Logical clock, C, st:

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

Allows us to order events, but still possible to receive messages out of order according to Lamport clocks – e.g., a sender with a "faster" clock vs. a sender with a "slower" clock, in logical time. We can guarantee we won't get an old message, only by polling all the nodes to see what their current time is (and advancing their time to my current time + 1).

Sometimes we can use application-level repair.
Whiteboard example, where we can optimistically draw any new message that arrives, as long as we have the ability to redraw. E.g:

Algorithm:

On message arrival:
if later than all previous messages, draw and add to end of list
else, insert into list at appropriate place, clear screen, and redraw from beginning (or from a checkpoint)

Paper uses example of mutual exclusion.
Correctness constraints:
one at a time
everyone in order of request

Liveness: (in absence of failures: if every lock holder eventually releases lock) eventually every requester will 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, and everyone's clock is > request)
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.

Illustrate algorithm 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.

Of course, doesn't work with failures, but we'll see how to generalize this to handle failures with paxos.

Sometimes we only need to avoid causality anomalies, without a total order. E.g., if an observer sees things that depend on things I've done, then they should also see the things I've done.

That's all that's needed for the lunch example – that no message that depends on a past event is delivered before that event.

A way of implementing this: 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).

On current process j, VC[j]+1 (for the local event of receiving the message)
For all i, VC[i] = max (VC[i], msg-VC[i])

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.

The set of reachable VC values are consistent states of the computation. Redraw previous example (sequence of RPCs) using vector clocks.
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

Various states are possible (valid): if an event is included, then all of the events that happen before it are included.

An application of this idea: consistent snapshot
[Chandy, Lamport. Consistent Snapshots. 1985]

"The … algorithm plays the role of a group of photographers observing a panoramic, dynamic scene, such as a sky filled with migrating birds-a scene so vast that it cannot be captured by a single photograph. The photographers must take several snapshots and piece the snapshots together to form a picture of the overall scene. The snapshots cannot all be taken at precisely the same instant because of synchronization problems. Furthermore, the photographers should not disturb the process that is being photographed; for instance, they cannot get all the birds in the heavens to remain motionless while the photographs are taken. Yet, the composite picture should be meaningful. The problem before us is to define "meaningful" and then to determine how the photographs should be taken."

Important for: distributed debugging, distributed checkpoints, distributed garbage collection, deadlock detection, distributed assertions, …

Reached here at the end of Wednesday

<another application of vector clocks: merging distributed log files>

For this we need some terminology:

Processes – sequence of events on one processor
Channels – in order sequence of packets sent from one process to another

A global state is the set of process/channel states, and a computation is a path through a sequence of global states.

A process can record its own state; the two processes at either end of a channel can cooperate to record the state of the channel. But we don't have a global real time clock, so we can't instantaneously record all processes and channels simultaneously. Rather, we need to use channels to coordinate the snapshot!

A consistent snapshot is one in which:
every process is checkpointed, and if an event happens before any event in a checkpointed process, that event is included in a checkpoint
every message sent by a process before its checkpoint is either received prior to a checkpoint on the receiver (and therefore included) or logged as part of the checkpoint of the channel.

For example, if we have a (one-way) channel between p and q:

Snapshot of p

Snapshot of q, before it has received any message sent after snapshot of p
All messages sent on channel before snapshot of p, and not received before snapshot of q

(Of course we have possibly many processes and channels.)

One possible algorithm: pick a time far in the future. Tell everyone about the time. When anyone reaches that time (in Lamport clock time), they do a local snapshot. (Also, snapshot any messages in flight at the time of the global snapshot.)
This is globally consistent! If an event happens before any event in a local snapshot, it is also included in the snapshot.

But suppose you want to do a snapshot soon?

Two rules:
Marker-Sending Rule for a Process p. For each channel c, incident on, and directed away from p:
p sends one marker along c after p records its state and before p sends further messages along c.

Marker-Receiving Rule for a Process q. On receiving a marker along a channel c:
if q has not recorded its state then
  begin q records its state;
        q records the state c as the empty sequence
  end
else
   q records the state of c as the sequence of messages received along c after q's state was recorded and before q received the marker along c.

E.g., for distributed garbage collection – any data structures that can't be reached in the consistent snapshot, are safe to reclaim. Data that is live in the consistent snapshot might be dead by the time we complete the snapshot! We're only guaranteed that data that was dead at the time the snapshot STARTED, will be seen as dead. Data that became dead during the snapshot might be seen as live or dead. Data that is live at the end, will be seen as live (or possibly, newly created).

But its ok to miss newly dead data, we'll take another snapshot later on, to collect new garbage.