Clock Synchronization and Coordination
1. Precision Inter
2. Accuracy

Physical clocks
if a -> b then C(a) < C(b)
if C(a) < C(b) then a -> b

NTP
A adjusts its time by \theta
where \theta = T3 - T4 + \delta
where \delta = (Treq + Tresp)/2
that means \theta = T3 - T4 + (((T2 - T1) + (T4 - T3)) / 2)
if \theta < 0 then A's clock is fast

To synchronize logical clock lamport defines a happened before relationship.

Happened before relationship:
1. If a and b are two events that occur in the same process, and a comes before b, then a -> b.
2. I a corresponds to sending a msg and b is the receipt of that message, then a -> b.
3. Transitive: a -> b and b -> c, then a -> b -> c

There is a partial ordering of events in a system.

There should be a message path between events to ensure their partial order.

If a !-> b then a || b (Concurrent)

This is the lamport algorithm to correct logical clock.

Lamport clocks are event counters
To implement Lamport's logical clocks, each process Pi maitains a local counter Ci. These counters are updated according to the following steps
1. Before executing an event (i.e. sending a message over the network, delivering a message to an application, or some other internal event), Pi increments Ci = Ci + 1
2. When process Pi sends a message m to process Pj, it sets m's timestamp ts(m) equal to Ci after having executed the previous step.
3. Upon the receipt of a message m, process Pj adjusts its own local counte ras Cj = max(Cj, ts(m)) after which it then executes the first step and delivers the message to the application.

The above algorithm is used to synchronize clocks.

if a -> b, then LC(a) < LC(b)
Lamport clock

if LC(a) < LC(b) but we cannot say a -> b.

If not(C(a) < C(b)) => not(a -> b) which means either b->a or a || b, that means there is no happened before relationship

Total ordering of timestamp
It's possible for Ci(e) == Cj(e)
i.e. Lamport clock timestamps are not totally ordered
We can break ties based on process-id
For process P_i, the clock value becomes Ci_i
for example for process 3 and counter 40, the LC becomes 40.3

An application of lamport clocks is Total order multicast.
Total order multicast is to ensure that the order of message received by the application are the same every where in the system.

Lamport clocks can be used to implement Total order multicast:
1. Consider a group of processes multicasting messages to each other. Each message is always timestamped with the current(logical) time of its sender. When a message is multicast, it is conceptually sent to the sender. In addition, we assume that messages from the same sender are received in the order they were sent, and that no messages are lost.
2. When a process receives a message, it is put into a local queue, ordered according to its timestamp. The receiver multicast the acknowledgement to the other processes. Note that if we follow Lamport's algorithms for adjusting the local clocks, the timestamp of the received message is lower than the timestamp of the acknowledgement.
3. A process can deliver a queued message to the application it is running only when that message is at the head of the queue and has been acknowledged by each other process. At that point, the message is removed from the queue and handled to the application.

Proof of lamport clock in total order multicast

Using Lamports clocks for Mutual exlusion.

Lamport clocks do not capture casuality. In practice, casuality is captured by means of vector clocks.

Vector Clocks

1. Before executing an event(sending a message over the network, delivering a message to an application, or some other internal event), Pi executes VCi[i] = VCi[i] + 1. This is equivalent to recording a new event that happened at Pi.

2. When process Pi sends a message m to Pj, its sets m's(vector) timestamp ts(m) equal to VCi after having exectued the previous step.(i.e, it also records the sending of the message as an evnt that takes place at Pi).

3. Upon the receipt of a message m, process Pj adjusts its own vector by setting VCj[k] = max(VCj[k], ts(m)[k]) for each k(which is equivalant to merging casual histories). after which it executes the first step and the delivers the message to the application.

Using Lamport clocks to achieve mutual exclusion
print Textbook page 328

Leader election

Chang-Robers leader election
First phase
1. to start an election, send your id clockwise as part of "election" message
2. id received id is greater than your own, send id clockwise
3. If receiced id is smaller, send your id clockwise
4. if received id is equal, then you are the leader
Second phase
1. Leader sends an "elected" message along with id
2. Other processes forward it and can leave the election phase.

lamport's mutual exclusion algorithm:
• Requesting the CS:
1. If P_i wants to enter the CS, it broadcasts a Request message (ts,i) and places the request on its own request queue
2. All processes place the request in their queue, ordered by timestamp, and send an ack to P_i
• Executing the CS: Process-i enters the CS when the following two conditions hold:
1. P-i has received a message with timestamp larger than ts from all processes
2. P-i’s request is at the head of the request queue
• Releasing the CS:
1. Remove request from queue and broadcast a timestamped Release message
2. When process-j receives a release message, it deletes P-i’s request from its queue

Correctness proof

• Proof by contradiction
• Suppose Pi and Pj enter the CS at the same time.
• This implies that at some point in time (t), both Pi and Pj had their own requests at the top of their respective queues
• Assume the timestamp of Pi is smaller than Pj . Recall that lamport timestamps can be totally ordered .
• This means that when P-i's request message was present in P-j's request queue, and P-j was already in the CS.
• But request queues are ordered by timestamps, and P-I's is smaller
• Assumes FIFO ordering of messages between proceses

Improvement to Lamports mutual exclusion algorithm

Quorum based

• Processes do not request permission from all other sites, but only a subset
• Every pair of processes has a processes that mediates conflicts between that pair
• Processes can send only one reply message at any time, and only after it has received a release message for the previous reply message
• Quorums must be mutually pairwise intersecting
• Quorums cannot contain complete subsets