FDS Assignment 1

G24AI1005
Abhijeet Kumar

Q1)
a) Using Lamport Clocks to ensure event order.

Let us assume that the ride hailing platform has following services.
① Rider Service - Handles rides requests
② Matching Service - Assigns a driver to the ride
③ Driver Service - Allows drivers to accept a ride
There can be other services as well, but for now, we will discuss with these three services.
Lamport timestamp implementation, requires ~~see~~ each service to maintain a logical clock. whenever a service recieves an event, ~~to~~ the service updates its logical clock using formula max(recieved timestamp, own counter)+1. And uses the new time stamp to publish the event.

Example flow -
Rider Service starts with counter 5 (from previous activities)
It recieves a ride request, ~~its increases~~ since ride request is a fresh request, it doesn't

have any logical time stamp yet.
So, rider service increments its logical clock to 6, and creates/publishes an event with timestamp 6.

Matching Service has counter 3 (from previous ~~[illegible]~~ clock)
Matching service recieves the event and updates its logical clock to value max(3,6)+1 which is 7. ~~the~~ Then matching service sends out request to riders with timestamp 7.

Riders recieve the request, they update their logical timestamps using formula max(~~to~~ recieved timestamp, own counter)+1}, and respond with an event with new timestamp.

The maching service then updates its timestamp and assigns rider to the requester.

This ensures that a logical flow of causality is implemented which can allow us to make critical desicions in events of concurrency & conflicts.

Example -

If 2 riders accept vide request from 1 user, then,

~~A accep~~

Rider A with timestamp 5 accepts request with timestamp 7, it updates its own counter to 8 and sends out the response to matching service,

Similary Rider B with timestamp 4, accepts request with timestamp 7, it updates its counter to 8 and sends out response to maching service with timestamp A.

lets assume that Rider A accepted the request first, ~~then~~ then maching service upon accepting the request, increases its counter to 9, and updates the event with timestamp 9.

When matching service recieves request from Rider B with timestamp 8, its own timestamp is 9 (after accepting Rider A's event). so matching service updates the event with time stamp max (8,9)+1 = 10.

This allows the matching service to conclude that Rider A should be given priority as Rider A was first to accept the ride. This is how lamport timestamps can help in events of concurrency.

In events of conflict, a tie breaker, a different algorithm to choose one, from the options, is used.

Using Vector Clocks for Causal Relationships

Vector Clocks requires each service to keep and maintain a vector to track logical timestamps of each service.

Using the same set of ~~servic~~ services as discussed above,

Rider requests a ride, request comes to rider service, with timestamp [5, 0, 0]
Rider service increments its counter and publishes event with new vector timestamp [6, 0, 0] and also updates its own logical vector clock.

Matching service recieves the event with timestamp [6,0,0]. Assuming matching servie has its own a vector timestamp of [0,3,0], it updates its own timestamp and ~~requests~~ creates an event for driver service with vector timestamp [6,4,0]

Driver recieves the event, and assuming its own timestamp is [0,0,2], it accepts the ride by sending event with new timestamp [6,4,3]

Advantages of vector clocks,

- They can reveal causality truely pointing out which service depend on ~~with~~ which servie (flow of events through services)
- They can ~~help~~ help distinguish dependent and independent events, for example, if ~~for~~ each timestamp V[i] in vector timestamp V is less than U[i] in vector timestamp U, always, then we can ~~conclu~~ conclude on dependency. If it is not a case always, the events are independent.
- Event ordering is more accurate compared to lamport timestamps. This can help with better conflict resolutions.

Implentation challenges will definitely arise ~~other~~ when there ~~to~~ are ~~ra~~ many services. It would then require much more memory and overhead compared to ~~ba~~ lamport time stamps.

c) Trade offs b/w lamport & vector clocks

Overhead - Lamport Clocks have low overhead compared to Vector clocks as vector clocks can be large when there are many services, ~~to~~ Managing and storing, as well as tagging events with such big timestamps will have large over head compared to lamport timestamps, ~~wh~~ where only one logical value is used. ~~wha~~

Scalability - lamport timestamps are highly scalable as they are fixed size timestamps unlike vector clocks, which grows with ~~num~~ number of services.

Ease of Deployment - Lamport timestamps are simple to implement compared to vector clocks.

Vector clocks are excellent when compared to lamport time stamps when it comes to precision in causality. But lamport timestamps are better in regards of overhead, scalability and implementation complexity.

Q2)

a) Ricart - Agrawala for drone scheduling.

lets assume, 3 drones D_1, D_2 and D_3 are used. Each of them maintains a logical clock, like lamport time stamp. (T_1)

Drone D1 wants to book a pick up time.
It sends requests to other drones with its own logical clock timestamp.

Other drones receive the request
if no drones are updating their schedule, they immediately send a REPLY message to D_1, allowing access.

If D_2 is already updating with timestamp T_2, then,

If $T_2 > T_1$, D_2 replies immediately allowing D_1 to proceed first

If $T_2 < T_1$, D_1 awaits (since D_2 has priority)

Once D_1 gets replies from all drones, it updates the schedule. After updating D_1 sends a RELEASE message, allowing other drones to proceed.

Concurrency control

The timestamps insure order, so no 2 drones update schedules at ~~the~~ the same time. Each drone must wait for permission from all others before proceeding.

~~Example~~ Flow

D_1 sends ~~Req~~ REQUEST(3, D1) to D2, B

D_2 is busy REQUEST(2, D2), D_1 waits

D_2 finishes and sends RELEASE - D_1 proceeds

D_1 updates and then sends RELEASE(3, D1)

b) Using token based approach
Instead of asking others for permission, wait for a single token, which grants access to update the schedule.

Using same set of drones,
D_1 wants to update the schedule
If D_1 has token, it can proceed
Else send request to token holder
If token is free, it is passed to D_1
D_1 updates the schedule and then passes the token to next requesting drone.

In large geographical setting
- The system keeps a distributed look up table (or a known drone acts as a token tracker)
- If a drone does not know who has the token, sends a broadcast request
- Drone holding the token, or token tracker replies

Flow

D_1 has token, updates the schedule and passes it to D_2

D_3 requests token from D_2, waits for D_2 to finish

D_2 finishes, passes token to D_3.

c) Handling drone / network failures

Ricart Agrawala

If D_1 crashes or there is network issue mid update, the drones D_2 and D_3 will wait until a timeout, and then they will continue.

Token Based,

If D_1 crashes while holding the token, or gets off network, then a new token will be generated using a consensus algorithm like (Paxos or Raft)

d) Best approach for high drone mobility and unstable connectivity.

Token based approach minimizes messaging overhead and simplifies failure

recovery and thus is more suitable for high mobility drones with ~~less less~~ intermittent connectivity.

Overhead

- Ricart Agrawala requires O(2N) messages (request + replies) to arrive at a conclusion which is inefficient for large fleet of high mobility drones.
- Token based - Usually O(1) messages are used for arriving at a conclusion.

Fault tolerance

- Ricart Agrawala - When one drone fails, others wait till timeout, causing long delays.
- Token based - If a drone without token fails, there is very less impact, however (chances are low) if a drone with token fails, a high overhead and delay of leader election will follow to regenerate the token. Once token is regenerated, usual flow will happen.