Northeastern University – Silicon Valley

CS 6650 Scalable Dist Systems

Homework Set #3 [100 points]

Name:Zheng Yin

email: yin.zheng@northeastern.edu

INSTRUCTIONS: ***Please provide clear explanations I your own sentences, directly answering the question, demonstrating your understanding of the question and its solution, in depth, with sufficient detail. Submit your solutions [PDF preferred]. Include your full name. Do not email the solutions.***

Study **Chapter 14 and 15 from** Coulouris Book

Answer the following questions using explanation and diagrams as needed. No implementation needed.

1. 14.2 [5 points]

The clock is essentially running ahead of the correct time by 4 seconds, so resetting it immediately to the correct time would mean that it would start losing time and would fall behind again. Instead, we need to adjust the clock such that it will run at the correct speed, compensating for the time it has gained so far.

To do this, we need to determine how much time the clock gains per second. Since the clock is 4 seconds fast and we want to correct it after 8 seconds have elapsed, we can calculate this by dividing the gain in time (4 seconds) by the elapsed time (8 seconds):

Time gain per second = 4 / 8 = 0.5 seconds

This means that the clock is gaining 0.5 seconds per second. To adjust it, we need to slow it down by that amount, so that it will keep correct time going forward. To do this, we need to let it run for 8 seconds while slowing it down by 0.5 seconds per second:

New elapsed time = 8 seconds
Time adjustment per second = -0.5 seconds

To adjust the clock, we can subtract the time adjustment from the actual time for each second that elapses:

Elapsed time	**Time adjustment**	**Adjusted time**
0 seconds	0.0 seconds	10:27:54.0
1 second	-0.5 seconds	10:27:53.5
2 seconds	-1.0 seconds	10:27:53.0
3 seconds	-1.5 seconds	10:27:52.5
4 seconds	-2.0 seconds	10:27:52.0
5 seconds	-2.5 seconds	10:27:51.5
6 seconds	-3.0 seconds	10:27:51.0
7 seconds	-3.5 seconds	10:27:50.5
8 seconds	-4.0 seconds	10:27:50.0

After 8 seconds have elapsed, the clock will be adjusted to 10:27:50.0, which is the correct time. By doing it this way, we ensure that the clock runs at the correct speed going forward and will keep accurate time in the future.

2. 14.4 [10 points]

To synchronize its clock with the time server, the client should use the minimum round-trip time of 20 ms as it is the fastest response received. Based on this, the client can estimate the current time by adding half of the round-trip time to the time reported by the server. This gives an estimated time of 10:54:28.352.

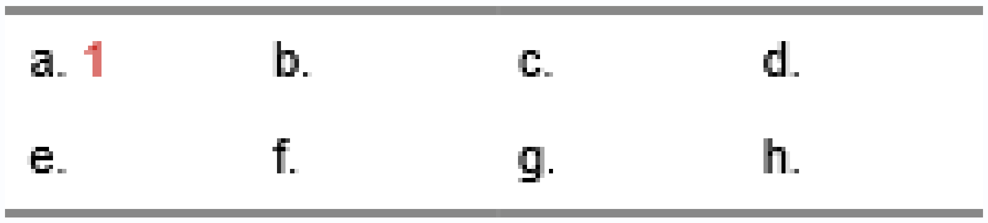
However, it is worth noting that using the minimum round-trip time may not always be the most accurate approach. In this case, the time difference between the fastest and slowest response times is only 5 ms, so using the average round-trip time would have given a more accurate estimate. Nonetheless, the estimated time of 10:54:28.352 is within ± 10 ms of the actual time reported by the server.

If the minimum message transfer time is known to be 8 ms, then the client should discard the first measurement with a round-trip time of 22 ms and use the second and third measurements instead. This would give an average time of 10:54:26.896, which is more accurate than the previous estimate. Using this approach, the estimated time would be accurate to within ± 2 ms of the actual time reported by the server.

3. A system of four processes, (P_1, P_2, P_3, P_4), performs the following events: [25 points]

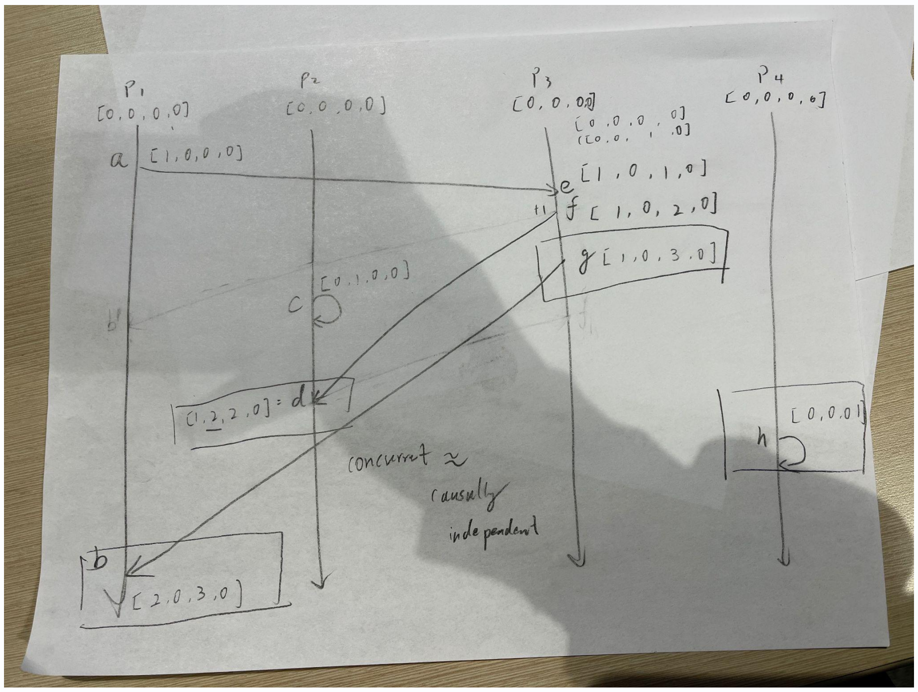
a. P_1 sends a message to P_3 (to event e).
b. P_1 receives a message from P_3 (from event g).
c. P_2 executes a local event.
d. P_2 receives a message from P_3 (from event f).
e. P_3 receives a message from P_1 (from event a).
f. P_3 sends a message to P_2 (to event d).
g. P_3 sends a message to P_1 (to event b).
h. P_4 executes a local event.

When taking place on the same processor, the events occur in the order listed.
Assign Lamport timestamps to each event. Assume that the clock on each processor is initialized to 0 and incremented before each event. For example, event *a* will be assigned a timestamp of 1.



a. 1	b.	c.	d.
e.	f.	g.	h.

a. Assign vector timestamps to each event in question 2. Assume that the vector clock on each processor is initialized to (0,0,0,0) with the elements corresponding to (P_1, P_2, P_3, P_4). For example, event *a* will be assigned a timestamp of (1, 0, 0, 0).



Lamport timestamps:
Event a: 1 (P1 sends a message to P3)
Event b: 5 (P3 sends a message to P1)
Event c: 1 (P2 executes a local event)
Event d: 4 (P2 receives a message from P3)
Event e: 2 (P3 receives a message from P1)
Event f: 3 (P2 receives a message from P3)
Event g: 4 (P1 receives a message from P3)
Event h: 1 (P4 executes a local event)

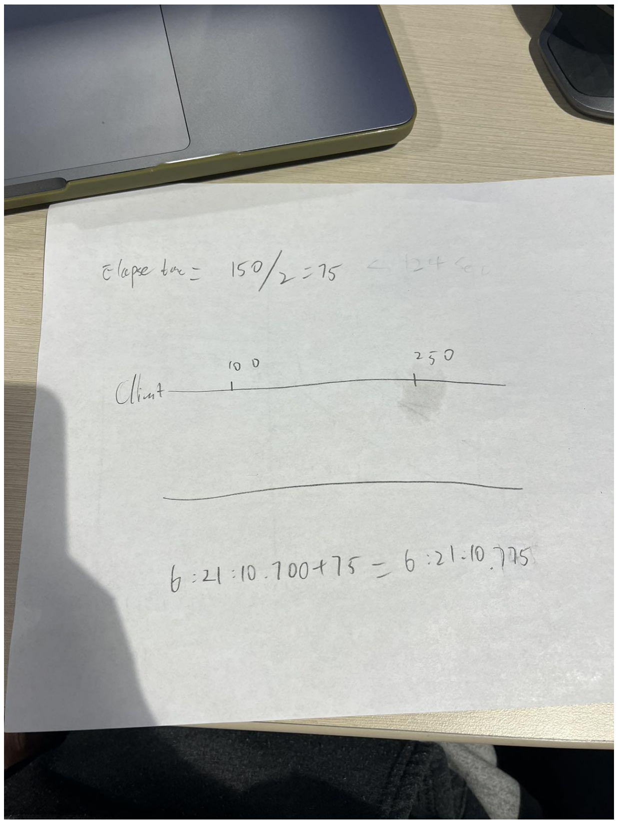
Vector timestamps:
Event a: (1, 0, 0, 0)
Event b: (2, 0, 3, 0)
Event c: (0, 1, 0, 0)
Event d: (1, 2, 2, 0)
Event e: (1, 0, 1, 0)
Event f: (1, 0, 2, 0)
Event g: (1, 0, 3, 0)
Event h: (0, 0, 0, 1)

b. Which events are concurrent with event *d*?

ANS: Event **b, g, h** are concurrent with event *d*

4. You are synchronizing your clock from a time server using Cristian's algorithm and observe the following times: [20 points]
    - timestamp at client when the message leaves the client: 6:22:15.100
    - timestamp generated by the server: 6:21:10.700
    - timestamp at client when the message is received at client: 6:22:15.250

▪ To what value do you set the client's clock?
▪ If the best-case *round-trip* message transit time is 124 msec (0.124 sec), what is the error of the clock on the client?



The round-trip message transit time can be calculated as the difference between the time the message left the client and the time it was received back at the client. In this case, it is:
Round-trip message transit time = (6:22:15.250 - 6:22:15.100) - (6:21:10.700 - 6:21:10.700) = 0.150 seconds
The offset between the client's clock and the server's clock can be calculated as half of the round-trip message transit time, since it takes the same amount of time for the message to travel from the server to the client and back:

Offset = 0.150 / 2 = 0.075 seconds.
To set the client's clock, we add the calculated offset to the server time:

Client's clock = 6:21:10.700 + 0.075 = 6:21:10.775
Therefore, the client's clock should be set to 6:21:10.775.

To calculate the error of the client's clock, we can compare the client's clock time with the actual time reported by the time server:

Error = 6:21:10.775 - 6:21:10.700 = 0.075 seconds

This means that the client's clock is accurate to within ±0.075 seconds of the server's clock. If the best-case round-trip message transit time is 0.124 seconds, then the error of the client's clock will be at most ±0.062 seconds, since the error is proportional to the round-trip time.

5. 15.1 [5 points]

It is possible to implement an unreliable failure detector using an unreliable communication channel. The unreliable failure detector is constructed in the same way as a reliable failure detector, with the only difference being that dropped messages may increase the number of false suspicions of process failure. Essentially, the unreliable failure detector assumes that a process has failed if it does not receive a heartbeat message from that process for a certain period of time. However, due to the unreliability of the communication channel, some heartbeat messages may be dropped, which can cause the failure detector to incorrectly suspect that a process has failed.

On the other hand, a reliable failure detector requires a synchronous system, and it cannot be built on an unreliable channel. This is because it is impossible to distinguish between a dropped message and a failed process, unless the unreliability of the channel can be masked while providing a guaranteed upper bound on message delivery times. In other words, the channel needs to ensure that messages will be delivered within a certain time frame, even if some of them may be dropped. For example, a channel that drops messages with some probability but guarantees that at least one message in a hundred will not be dropped could be used to create a reliable failure detector.

In summary, while it is possible to implement an unreliable failure detector using an unreliable channel, building a reliable failure detector on an unreliable channel is challenging, and requires careful consideration of the message delivery times and the probability of message loss.

6. 15.3 [10 points]

If s = synchronization delay and m = minimum time spent in a critical section by any process, then the maximum throughput is 1⁄(sm+) critical-section-entries per second.

7. 15.5 [10 points]

The server uses a reliable fault detector to determine if any of the clients have crashed. If the client has been granted a token, the server acts as if the client has returned the token. If it subsequently receives a token from a client (possibly sent before the crash), it ignores it.

The resulting system is not fault-tolerant. If a token-holding client crashes, application-specific data protected by critical sections (whose consistency is at stake) may be in an unknown state when another client starts accessing it.

If the token-holding client is incorrectly suspected of failing, then it is possible that two processes are allowed to execute in the critical section at the same time.

8. 15.20 [15 points]

To RTO-multicast a message, a process attaches a unique identifier to it and sends it to all other processes using R-multicast. Each process keeps track of the messages it has received and the messages that have been totally-ordered (RTO-delivered). When a process wants to deliver the next set of messages, it proposes its set of not-yet-RTO-delivered messages to the other processes. This proposal is collected by the other processes and a consensus is reached on which messages should be delivered next.

When a process receives the consensus decision, it takes the intersection of the decision value and its set of not-yet-RTO-delivered messages and delivers them in the order of their identifiers, moving them to the record of messages it has RTO-delivered. This ensures that every process delivers messages in the order of the concatenation of the sequence of consensus results, which guarantees a RTO multicast.

In summary, the RTO-multicast protocol uses a combination of unique identifiers, R-multicast, consensus algorithm, and intersection to ensure that messages are delivered reliably and in a totally-ordered manner.