CS 7610: HW-1

Name: Sumeet Mahendra Gajjar

Solution 1.a:

This algorithm is an extension of Cristian's Algorithm to synchronize clocks of n processes. Any process can initiate the algorithm and each process acts as a server and as a client and runs the following algorithm.

Algorithm 1 Algorithm to synchronize clocks of n processes

```
1: A process initiates the algorithm by broadcasting a "start" message, followed by broadcasting its time to all its peers.
2: Each process on receving the "start" message broadcasts their time to all the peers.
3: $T_{peer-i}$ indicates the time of $ith$ peer.
4: $T_{recv-i}$ indicates the time at which the current process received the time from the $ith$ peer.
5: $RTT_i$ indicates the Round trip time from current process to $ith$ peer.
6: $A = \left\{\forall i : T_{comp-i} \leftarrow \left(T_{peer-i} + \frac{RTT_i}{2}\right) : T_{comp-i} > T_{recv-i}\right\}$    ▷ selecting peers which are ahead
7: if $A \neq \emptyset$ then
8:     $T_{current-process} \leftarrow \max(A)$
9: else
10:    No modification required since the current process's clock is ahead of everyone.
11: end if
```

Solution 1.b:

Due to the uniform nature of the RTT, it surely impacts the processes in a negative manner and it can be clearly seen during the cases when RTT is at its peak value and the clock synchronization begins. As stated in the Cristian's algorithm, the client time is calculated as: $T_{client} \leftarrow T_{server} + \frac{RTT}{2}$. It can be clearly seen that if the recorded RTT value is at its peak, then the client will jump ahead of the server, their clocks will not be in sync. This defeats the purpose of using the algorithm in the first place.

Improvements:

- Instead of querying the server once, query the server n times and take the average of n RTTs to lower the impact.
- While querying the server, if the RTT is beyond a certain threshold ϵ, ignore the measurement altogether to avoid skewing the average.
- To further reduce the error, client can use "on the wire protocol" to calculate RTT. This removes the processing time taken by the server from the equation and results into a more accurate value.

Solution 2.a:

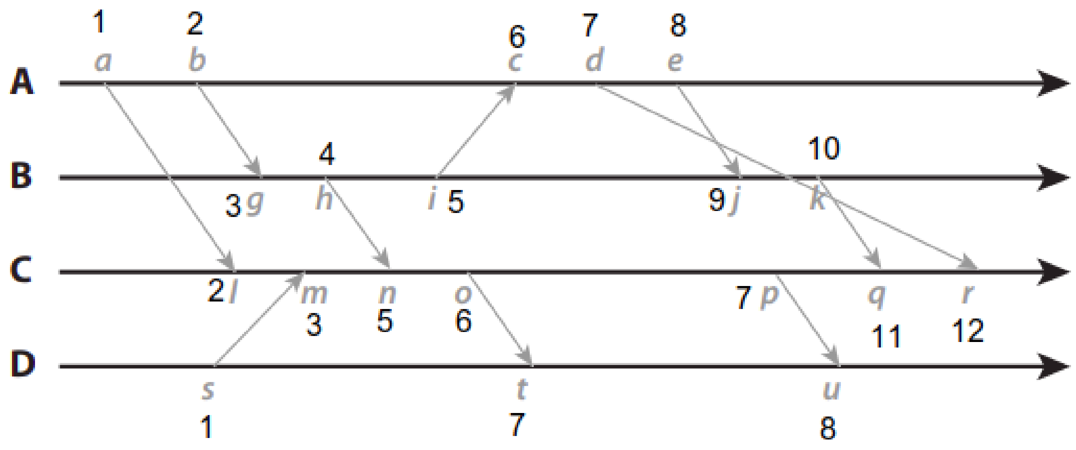

Figure 1: Lamport clocks

event	a	b	c	d	e	g	h	i	j	k
clock	1	2	6	7	8	3	4	5	9	10
event	l	m	n	o	p	q	r	s	t	u
clock	2	3	5	6	7	11	12	1	7	8

Solution 2.b:

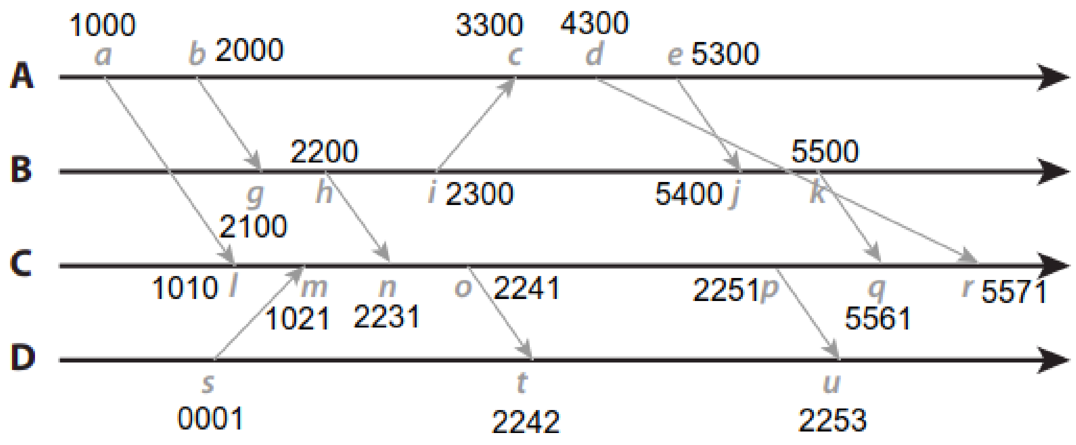

Figure 2: Vector clocks

event	a	b	c	d	e	g	h	i	j	k
clock	1000	2000	3300	4300	5300	2100	2200	2300	5400	5500
event	l	m	n	o	p	q	r	s	t	u
clock	1010	1021	2231	2241	2251	5561	5571	0001	2242	2253

Solution 3:

A relation R is over a set S is total order if the following statements hold for all distinct a, b and c in S:

1. Antisymmetric: if aRb with $a \neq b$ then bRa must not hold
2. Transitive: if aRb and bRc then aRc
3. Connexity: if aRb or bRa

Lamport clocks defines C_i as a counter which is maintained by a process p_i. $C_i(a)$ is a function which provides the value of the counter at a event a. It defines a relation $\rightarrow$, such that for any events a and b at a process p_i

$$\text{if } a \rightarrow b \text{ then } C_i(a) < C_i(b)$$

Let's consider distinct events $a, band c$ of process p_i.

- Antisymmetric: As per the definition of C_i, since a and b are distinct events and time always moves forward(at least in this universe), $a \rightarrow b$ holds or $b \rightarrow a$ holds, but both cannot be true. Thus the Antisymmetric property is satisfied.
- Transitive: For $a \rightarrow b$ and $b \rightarrow c$, as per the definition, $C_i(a) < C_i(b)$ and $C_i(b) < C_i(c)$ respectively. Thus $C_i(a) < C_i(c)$, which means $a \rightarrow c$, which satisfies the transitivity property.
- Connexity: If the events a and b belong to the same process, then by definition of $a \rightarrow b$, $C_i(a) < C_i(b)$ holds true. The only time value of $C_i(a)$ and $C_i(b)$ can be equal is when a and b belongs to two different process. In such cases the tie can be broken by using the process identifiers. Thus if a is event in p_i and b is event in p_j, then $a \rightarrow b$ iff $C_i(a) < C_i(b)$ or $C_i(a) = C_i(b)$ and $p_i < p_j$. Hence the relation satisfies Connexity property.

Since all above properties are satisfied, Lamport clocks algorithm provides a total order.

Solution 4:

P0 saves its state before sending marker messages and starts recording on incoming channels $P1 \rightarrow P0$ and $P2 \rightarrow P0$

Node	Recorded State						
	State	$P1 \rightarrow P0$	$P2 \rightarrow P0$	$P0 \rightarrow P1$	$P2 \rightarrow P1$	$P0 \rightarrow P2$	$P1 \rightarrow P2$
P0	{}	{}	{}				
P1							
P2							

P1 receives the marker message from P0, it saves its local state, records $P0 \rightarrow P1$ as empty and starts recording on incoming channel $P2 \rightarrow P1$.

Node	Recorded State						
	State	$P1 \rightarrow P0$	$P2 \rightarrow P0$	$P0 \rightarrow P1$	$P2 \rightarrow P1$	$P0 \rightarrow P2$	$P1 \rightarrow P2$
P0	{}	{}	{}				
P1	{B}			$\emptyset$	{}		
P2							

P0 receives the marker message from P1.

Node	Recorded State						
	State	$P1 \rightarrow P0$	$P2 \rightarrow P0$	$P0 \rightarrow P1$	$P2 \rightarrow P1$	$P0 \rightarrow P2$	$P1 \rightarrow P2$
P0	{}	{A}	{}				
P1	{B}			$\emptyset$	{}		
P2							

P2 receives the marker message from P1, it saves its local state, records $P1 \rightarrow P2$ as empty and starts recording on incoming channel $P0 \rightarrow P2$

Node	Recorded State						
	State	$P1 \rightarrow P0$	$P2 \rightarrow P0$	$P0 \rightarrow P1$	$P2 \rightarrow P1$	$P0 \rightarrow P2$	$P1 \rightarrow P2$
P0	{}	{A}	{}				
P1	{B}			$\emptyset$	{}		
P2	{}					{}	$\emptyset$

P2 receives the marker message from P0, it terminates the algorithm since it received markers on all its incoming channels.

Node	Recorded State						
	State	$P1 \rightarrow P0$	$P2 \rightarrow P0$	$P0 \rightarrow P1$	$P2 \rightarrow P1$	$P0 \rightarrow P2$	$P1 \rightarrow P2$
P0	{}	{A}	{}				
P1	{B}			$\emptyset$	{}		
P2	{}					{}	$\emptyset$

P0 receives the marker message from P2, it terminates the algorithm since it received markers on all its incoming channels.

Node	**Recorded State**						
	State	$P1 \rightarrow P0$	$P2 \rightarrow P0$	$P0 \rightarrow P1$	$P2 \rightarrow P1$	$P0 \rightarrow P2$	$P1 \rightarrow P2$
P0	{}	{A}	{D}				
P1	{B}			$\emptyset$	{}		
P2	{}					{}	$\emptyset$

P1 receives the marker message from P2, it terminates the algorithm since it received markers on all its incoming channels.

Node	**Recorded State**						
	State	$P1 \rightarrow P0$	$P2 \rightarrow P0$	$P0 \rightarrow P1$	$P2 \rightarrow P1$	$P0 \rightarrow P2$	$P1 \rightarrow P2$
P0	{}	{A}	{D}				
P1	{B}			$\emptyset$	{F}		
P2	{}					{}	$\emptyset$

Following table indicates the final local and channel states after algorithm termination:

Node	**Recorded State**						
	State	$P1 \rightarrow P0$	$P2 \rightarrow P0$	$P0 \rightarrow P1$	$P2 \rightarrow P1$	$P0 \rightarrow P2$	$P1 \rightarrow P2$
P0	{}	{A}	{D}				
P1	{B}			$\emptyset$	{F}		
P2	{}					{}	$\emptyset$

Solution 5:

Chandy-Lamport snapshot algorithm assumes the communication is FIFO across processes. This means if a process sends a marker message and then sends a message A, the marker message will be delivered before message A.

Given the scenario where the communication is not FIFO, Chandy-Lamport algorithm cannot be used since the recorded global state won't be consistent. In such cases, we can use snapshot algorithm proposed by Friedemann Mattern in *"Virtual Time and Global States of Distributed Systems"*.

In a nutshell, this algorithm mimics a real world snapshot algorithm, where every party agrees on a future time "s" at which a snapshot should be taken and at time "s" all parties take a local snapshot. A global snapshot is constructed out of all local snapshots.

The real-world algorithm assumes all parties refer to a global clock, however, this assumption cannot be true in Distributed Systems. So for systems without a global-clock, a *virtual-time* is used and in this case, we use Vector clocks. For simplicity, we assume there is only one snapshot request initiator P_i and processes do not crash during the snapshot algorithm.

Mattern's Algorithm:

1. P_i "ticks" and fixes a future time $s = C_i + (0, ...0, 1, 0, ...0)$ as the common snapshot time. Here C_i is the vector clock of P_i and the "1" is present at position i. It broadcasts "s" to all other processes and waits for the acknowledgements from all its peers.

2. On receiving "s" from P_i, all peers store "s" and sends back the acknowledgement to P_i.

3. On receiving acknowledgements from all its peers, P_i "ticks" again, this sets C_i to "s". It now takes a local snapshot and broadcasts a dummy message to all processes.

4. On receiving the dummy message, all peers advance their clocks to a value $\geq s$

5. As soon as the clock value on peers becomes $\geq s$, all peers takes a local snapshot, thus a peer can take a local snapshot before the dummy message has arrived. Once the snapshot is captured, it sends it to P_i.

6. The state of C_{ij} is all messages sent along C_{ij}, whose timestamp is smaller than s and which are received by P_j after recoding LS_j. Here LS_j means local snapshot at P_j.

7. In order to terminate the algorithm, a termination detection scheme for non-FIFO channels is required.

Termination detection Algorithm:

A process can have either of the two states; White state: before taking snapshot and Red state: after taking snapshot. White processes send white message and Red processes send red messages. Each process is initially white and turns red immediately after taking a local snapshot.

1. $\forall i : P_i$ keeps a $counter_i$ that indicates the difference between the number of white messages sent and received before recording its LS_i.
2. This $counter_i$ is reported to the initiator process along with LS_i and it forwards all the white messages it receives after taking the snapshot to the initiator.
3. The algorithm terminates when the initiator has received $\sum_i counter_i$ forwarded white messages.

Correctness: Before proving the algorithm correct, let's first define when will a global state be consistent.

A global state GS is a consistent state iff it satisfies the following conditions:

- C1: $send(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus recv(m_{ij}) \in LS_j$, where $\oplus$ is XOR.
- C2: $send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge recv(m_{ij}) \notin LS_j$

Let's assume that a message m_{ij} is sent from process P_i to process P_j after process P_i records its LS_i. The message m_{ij} will have a timestamp $> s$ since the message was sent after recording snapshot.

Let's assume m_{ij} is received by process P_j before it records LS_j. On receiving m_{ij}, the clock of P_j reads a value $C_j > s$. According to step 5 of Mattern's algorithm, P_j should have recorded the LS_j which contradicts our assumption that m_{ij} was received before recording snapshot. Hence our assumption is wrong and m_{ij} cannot be received by P_j before it records LS_j. According to step 6 of Mattern's algorithm, m_{ij} will not be recorded in the SC_{ij} and thus C1 is satisfied.

$\forall m_{ij}$, whose timestamp is $< s$ and are received by P_j before taking the snapshot are included in LS_j. If they are received after taking the snapshot, m_{ij} will be included in the state of channel C_{ij}. This satisfies C2.

Thus we can say the snapshot algorithm is correct.