Question 1:

In this distributed system, computer C must be able to deduce the relationship between events. Events occurring at B would happen for sure only if B received a fire signal from A, in which case A 'happens before' B (A -> B) since as soon as A sends a message to B, B activates sprinklers to try to put down the fire. There are some cases when C receives a message from A after B even if A for sure preceded B, like the example of Alice and Bob in the ticket office.

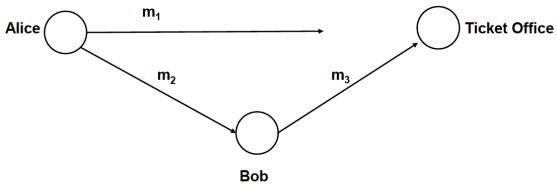

Figure 1: Example from handouts

Computer C needs to deduce what message arrives before and what message arrives after, in case we want a message to be put aside knowing that there is another message preceding the one just received.

We know that piggybacking all history is not efficient, so we need to implement timestamps. The system is flawed because it incorrectly enforces an erroneous happens before relationship if the fire station detects a message of new fire after another previous fire that surpasses a fire out message from B.

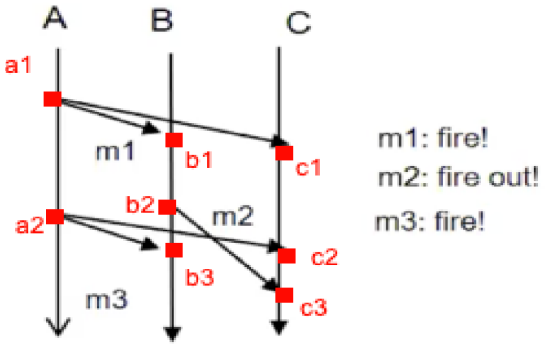

Figure 2: Highlights of events

In this case, if Computer C receives a message of fire out within 2d+e time after two consecutive messages of fire, it assumes that both have been put out and it won't send the fire truck. A sends a message to B and to C at the same time (m1). The fire is put off from sprinklers, so B sends a message (m2) to C informing that the fire brigade is not necessary, but the message is somehow delayed. Meanwhile, a new fire arises, and A sends a new message to B and to C (m3). Now after this message, C receives the old message m2 from B, so the system thinks that the fire brigade is not necessary even if there is a new fire.

We must tell the system that m2 has been sent before m3 and that the two messages are concurrent so there is a new fire. In this way, Computer C can understand that only the first fire had been put out but that another fire has started.

To deduce the right order of events, so that m2 and m3 were not sent at the same time and that m3 did not happen before m2, the messages must be concurrent because both messages appear at the same time in the diagram and m2 is not responsible for m3 and vice versa m3 is not responsible for m2. There is no happen relationship between them, m3 is concurrent to m2 so ~~m3 > m2~~ and ~~m2 > m3.~~ This is because the message m3 is coming from a new fire alarm.

Since events (messages) from B can be concurrent to events of A, m2 must be ordered before (=> total order) m3. If doing so C can recognise which one has higher priority looking at their timestamp (lower timestamp means that A precedes B, so it has higher priority.)

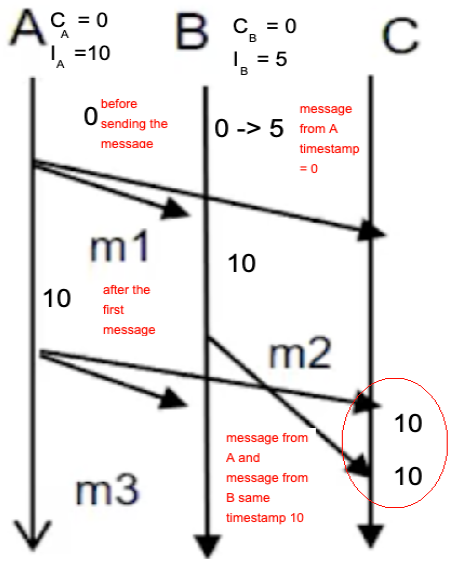

Figure 3: concurrent messages

As we can see concurrent events might also happen at the same time, so C has to recognise priority because of the timestamp

using the rule of breaking ties when they are concurrent and one ordered before the other. This can be implemented requiring Computer A's clock to hold a greater value than Computer B clock, it must be incrementing at least twice the value.
To do so, we can form a sequence of events knowing that event a occurs before event b. Computer B starts sprinklers only after receiving a message from Computer A so if 'a' is sending of a message by one process and 'b' is the receipt of the same message by another process, then a->b.
We have to be sure that messages from B (since they are coming only if a message has arrived from A, if a-> b then Clock of a < Clock of b) hold a greater time value than the time of the message arrived from A.
Since all messages contains a time and each process increments the clock after the occurrence of a local event, the system need to assign clocks when receiving and sending messages as events.
Consider the situation when all of the three computers start their clocks at zero 0. The incrementation of A must be however greater than the incrementation of B to allow a message from B to be sent to C to say that the fire is off before another message can be sent from A to C to say that the fire is still on. The timestamp still distinguishes between the two fires such as in the event of a message from A arriving to C successive and concurrent of a message from B arriving to C in mixed order it is possible to define which one arrives before.

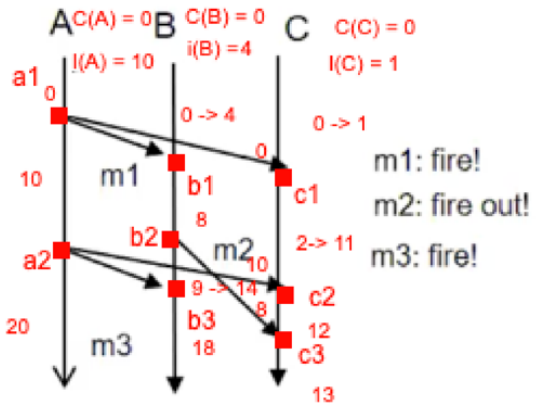

Figure 4: Correct behaviour when both timestamps start from same value.

Assuming at most one message can be sent from B after the arrival at B of a message from A, the increment of A must be greater than the increment of B.
In this example the timestamp of m2 is 8, while the timestamp of m3 is 10.

a1 -> b1 0 < 4	a1 -> c1 0 < 1
a2 -> b3 10 < 14	a2 -> c2 10 < 11
b2 || a2 8 < 10	b2 -> c3 8 < 12

A message coming from B to computer C can be both concurrent (//) with a new message from A if the message from B relates to a previous fire and have a happen before relation (->) if the fire is out (message from B) after receiving a fire! message from A.
The system must be implemented in the way that a message coming from Computer B holds a greater timestamp value of the message coming from A which has a happen before relation with but lower timestamp of the message from A which is concurrent to.
If C receives a message from Computer A with a smaller timestamp of the message received from B regardless of the order, C knows that the fire! received before are off while if the timestamp is greater, it relates to a new fire.
If the initial value of A is greater or equal than the initial value of B (e.g. both starting at 0,) the incrementation of A must be greater than the twice the incrementation of B.
If C(B) <= C(A) : I(A) > 2* I(B)
This prevents C from elaborating a new message m3 from A before the message from B m2 sent after reading the previous A message m1.
Since the initial value of B is not greater than the initial value of A, B will 'peek' the timestamp from m1 and increase it with its incrementation value. That's why in this case of C(A) >= C(B) the initial value of A plus the incrementation of A must be greater than the incrementation of B * 2 plus the initial value of A because B 'peaks' the timestamp from A message m1.
Then eventually if the fire is out off, B send a new message. There always are at most two incrementations to consider (for Implementation Rule 1 IR1), event of B reading the message and the event of B sending a message of fire off. Two events because B only activates after A for sure, B must advance his clock after reading the message value.

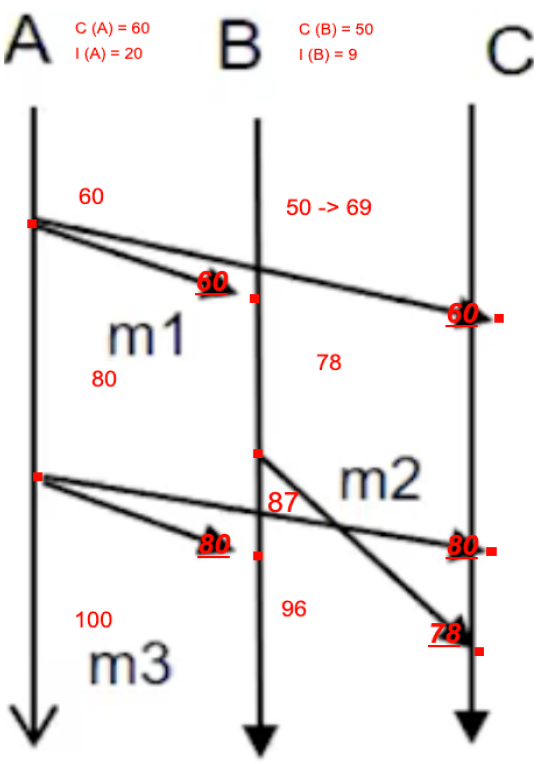

Figure 5: Correct behaviour when starting timestamp C(A) is greater than C(B)

a1 -> b1 60 < 69

b2 || a2 78 < 80 (m2, m3)

In case the initial value of A C(A) is lower than the initial value of B C(B), the sum between the initial value of A and the incrementation of A I(A) must be greater than the sum of the initial value of B C(B) and the incrementation of B I(B).

If C(B) > C(A) : [C(A) + I(A)] > [C(B) + I(B)]

A must be bigger than B, possibly by a fraction so that if C receives a message from B (A -> B) after receiving a message from A which has been sent before from a larger timestamp, C can determine whether the fire has been put out. What is really important is that the system can distinguish between a happen before relationship (a->b) when B sends a fire-out message m2 to C after receiving a message of fire m1 from A and when processes are concurrent (a//b) so the message m2 from B is related to a previous message m1 from A, even if Computer C for some delays receives it after a new message m3 from A. Since the system obeys to IR1 and A can send messages only to B and C at the same time and B can only send to C while C can only receive, incrementation of A must be greater than the incrementation of B.

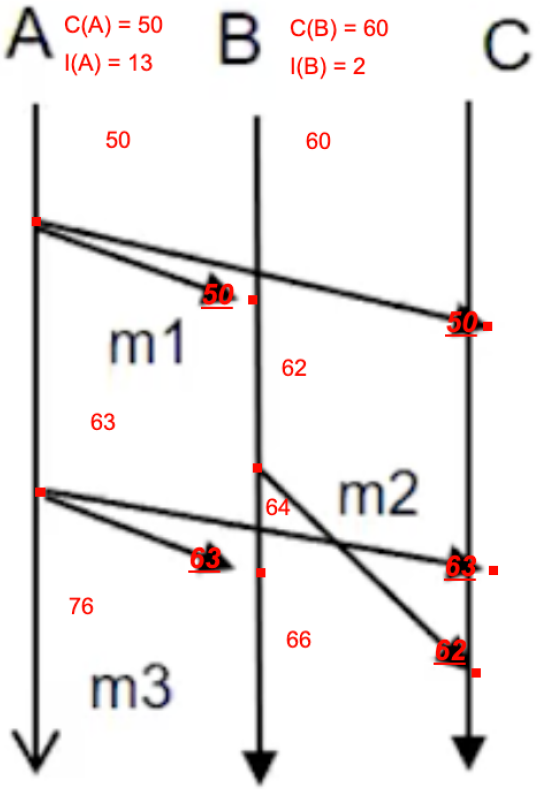

Figure 6: Correct behaviour when starting timestamp C(B) is greater than C(A)

a1 -> b1 50 < 60 b2 || a2 62 < 63

These diagrams represent the event of Computer A detecting a fire and sending a message to both B and C. Then Computer C starts a timer that is of 2d+e units of time to wait a fire out message from Computer B. Before processing the message, computer B peeks its timestamp and if it is lower, it increments it accordingly to be greater than the one from the incoming message. If the fire is put off in a time less than 2d+e (otherwise the fire brigade has been sent) B sends a message to C (it takes d time.) Another fire starts and A sends a message to B and C again. C receives this message before the message sent from Computer B. Since we implemented timestamp correctly, Computer C is able to deduce that the latest arrived message from B refers to the first fire alert from A (breaking the Implementation Rule 2 IR2 since the new message has a timestamp with a lower value than the previous message sent to C.) This permits C to set a new timer and waits for a new eventual message of fire off from B before dispatching a fire truck (if the fire has been put out a truck does not need to be sent.) This way computer C can correctly assume how many fire are occurring and how

many have been put off in the 2d+e time since all events have timestamps correctly ordered the one sent after another message will hold a greater timestamp than the previous message, if not Computer C will be able to deduce that the message is concurrent with another. If Computer C keeps receiving new messages with increasing timestamps, it knows that there are new fires, and a new timer (2d+e units) needs to be set (or the fire has been put out if the message comes from B.) The timer considers both d that is the maximum time for a message to arrive to another computer and e that represents the maximum time that can elapse between sprinklers being activate by computer B and the fire being off. Therefore, even from the previous example, messages from A takes 0 time to arrive to C while messages from B take maximum time d and sensors take maximum time e to inform about a fire off, Computer C is always informed at the right time 2d+e being the shortest time possible that might pass before the system makes a definitive choice on sending fire trucks.

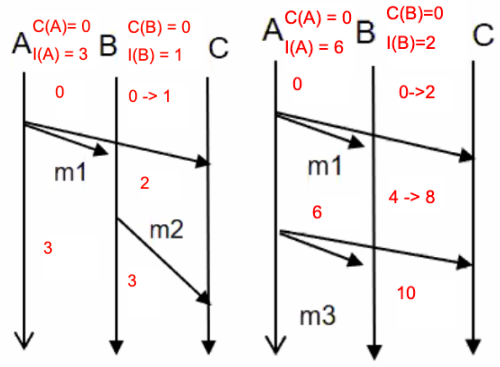

Figure 7: Correct behaviour for figures a and b with incrementation of A being greater than the double of the incrementation of B

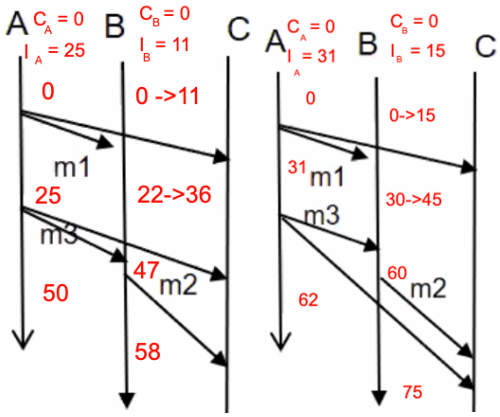

Figure 8: Correct behaviour for figures d and e

Question 2:

If the system uses physical clocks for timestamping which are synchronised to be within a known error bound Ɛ, any message sent is received within a maximum period of d time units and d is known. Fire engines are sent only if no fire out message goes from B to C within 2d+e time after a fire message has been received from C. If physical clocks instead of logical clocks are used, the performance of the system would be regulated by the synchronisation using Ɛ that must be as small as possible.

$\varepsilon < (1-k)\,\mu$

Where μ is the minimum time, it would take for a message to be sent from A and B. Any message is received within a maximum period of d time units and d is known.

The transmission time is between 0 and d, so the minimum time is zero. Using the formula above, Ɛ can't be close to zero or smaller than 0 (considering k equal to 0 and μ close to zero,) because perfect synchronised clocks are only ideal, it is not possible to show how the design in part (I) holds for some non-zero value of epsilon.

The requirements of (I) would not hold anymore because the only events that advance clocks are sending or receiving messages (point 3) and (point 4) since we would not use logical clocks anymore but physical clocks which advance at the passage of a reference (real) time.

Point 3 is not correct anymore because the physical clocks need to increment continuously as long as the time passes and not only when receiving or sending a message. Point 4 also conflicts since the values that logical clocks hold as timestamp are not needed anymore if physical clocks 'tick' with a reference time, so the system doesn't need to choose an increment value for the logical clock.

Since the system ordered events totally, it cannot guarantee to respect the ordering of events based on real time and this can give rise to anomalous behaviour.

Computer C might not be able to identify the difference between timestamps of a fire message coming from A and a fire-out! message coming from B sent immediately after receiving the message alert from A with a minimum transmission time close to zero and e time close to zero so fire put off by sprinklers immediately.

Similarly, Computer C is not able to distinguish the order of messages about multiple fires starting and being distinguished within μ a very short timeframe. This happens because with physical clocks we don't have the same incrementation as logical clocks therefore events might hold timestamps that are almost the same, very close to each other.

Computer C receiving messages with subsequent timestamps is not able to detect that a new fire is occurring immediately after a previous one has been extinguished. For example, A sends a message to C "Fire!" than B sends a message to C with the same timestamp number since it occurred immediately after it. So, there would be issues at recognising multiple messages with values very close each other leading to an anomalous behaviour from the system.

To meet the requirements (I) the system continuously needs to increment clocks as time passes without needing the timestamp incrementation values anymore; for this reason, the rules 3 and 4 must change.