2. Time, Clocks and Ordering of Events

T1

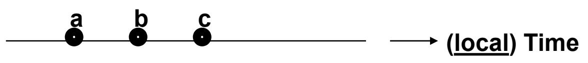
- In a distributed system, it is often necessary to be able to establish relationships between events occurring at different processes:
  - was event ‘a’ at P_1 responsible for *causing* ‘b’ at P_2?
  - is event ‘a’ at P_1 unrelated to ‘b’ at P_2?
- For event ‘a’ to have caused event ‘b’, ‘a’ must have *happened before* ‘b’
  - Not necessarily the other way round! Why?
  - ‘a’ *happened before* ‘b’ implies *potential* causality of ‘a’ on ‘b’
- We will discuss the partial ordering relation “happened before” defined over a set of events.

Assumptions

T2

(i) Processes communicate only via messages

(ii) Events of a process form a sequence, where event ‘a’ occurs before ‘b’ in this sequence if ‘a’ happens before ‘b’ (Note: events of a process are totally ordered):



(iii) Sending or receiving a message is an event

- We define the ‘happened before’ relation, denoted by ‘$\rightarrow$’ as follows:

Definitions

T3

- Definition 1: The relation '$\rightarrow$' on the set of events of a system satisfies the following three conditions:

  (i) if 'a' and 'b' are events in the same process, and 'a' comes before 'b' then $a \rightarrow b$

  (ii) if 'a' is sending of a message by one process and 'b' is the receipt of the same message by another process, then $a \rightarrow b$

  (iii) if $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$

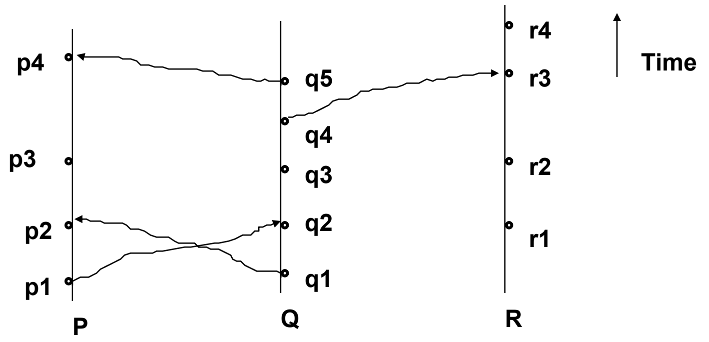
  Being meaningful, we assume $a \neg\rightarrow a$// irreflexive

- Definition 2: Two distinct events 'a' and 'b' are said to be concurrent (a || b) if $a \neg\rightarrow b$ and $b \neg\rightarrow a$.
- So, $\rightarrow$ defines a partial order over the set of events

  Since there could be concurrent events in the set, that by definition are not related by $\rightarrow$.
- Another way of regarding '$a \rightarrow b$' is to say that 'a' can have causal effect on 'b'

Space time diagram

T4



$a \rightarrow b$: one can go from a to b in the diagram moving forward in time along the process and message lines.

$p1 \rightarrow r4$, $q4 \rightarrow r3$, $p2 \rightarrow p4$, $q3 \parallel p3$, $q3 \parallel r2$

Logical Clocks

T5

- Problem: How to implement an event numbering scheme which respects the ' $\rightarrow$' relation. We do this by using logical clocks.

- Logical Clocks:

  We introduce clocks into the system . We begin with an abstract point of view in which a clock is just a way of assigning a number to an event.

  We define a clock C_i for each process P_i to be a function which assigns a number $C_i(a)$ to 'a' in P_i.

  The entire system of clocks is represented by the function C which assigns to any event 'b' the number C(b), where $C(b) = C_j(b)$ if 'b' is an event in P_j.

- How to specify the correctness condition for these clocks?

  Since our clocks are not measuring the passage of time, we cannot base our condition on physical time. We use the relation $\rightarrow$ for stating the correctness condition:

- <u>Clock Condition</u>:

  for any events a, b: if $a \rightarrow b$ then $C(a) < C(b)$

Logical Clock Conditions

T6

- Note that we cannot expect the converse of the clock condition to hold as well

  we cannot say that if $C(a) < C(b)$ then $a \rightarrow b$, since this would require two concurrent events to occur at the same logical time (have the same clock value).

- The clock condition can be satisfied if the following two conditions hold:

  CL1: if a and b are events in P_i and $a \rightarrow b$, then $C_i(a) < C_i(b)$.

  CL2: if a is the sending of a message by P_i and b is the receipt of that message by P_j, then $C_i(a) < C_j(b)$.

Implementing logical Clocks

T7

- **Implementation Requirement:**

  Each process (P_i) has a register (C_i) and $C_i(a)$ is the value contained in C_i during event a.

- **<u>Implementation Rules</u>:**

  <u>IR1</u>: each process P_i increments C_i immediately after the occurrence of a local event

  - **IR1 meets the condition CL1. To meet condition CL2, we require that each message contain a clock value, Tm (called the timestamp)**

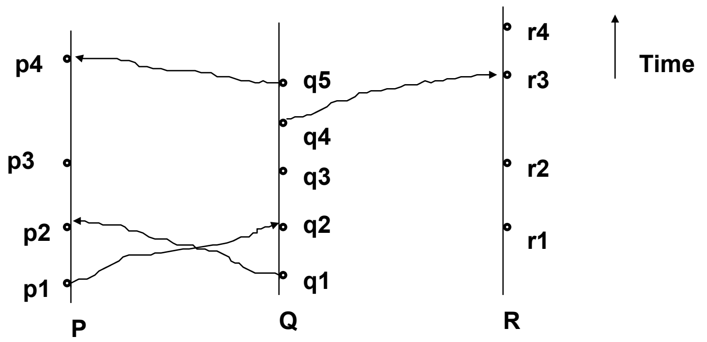
  <u>IR2</u>: (i) if 'a' is an event representing the sending of a message 'm' by P_i to P_j, then m contains the the timestamp Tm = $C_i(a)$

  (ii) receiving m (event b) by process P_j : peek at m; increments C_j, if necessary, to ensure that $C_j(b) > Tm$; execute receive(m).

Space time diagram

T8

- **A way of understanding *happened-before* (or lack of it)**

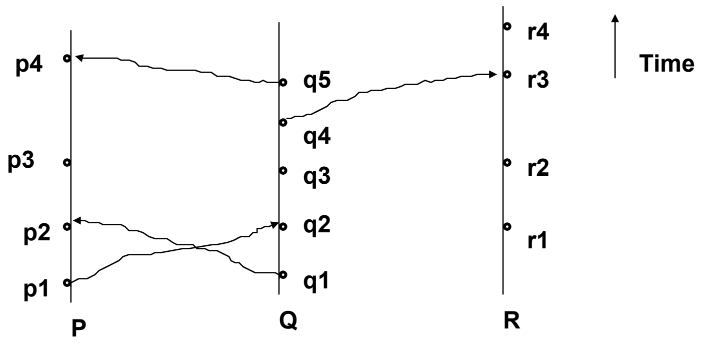


a → b : one can go from a to b in the diagram moving forward in time along the process and message lines.

p1 → r4, q4 → r3, p2 → p4, q3 || p3, q3 || r2

Space time diagram - Exercise

- Experiment with different initial clock values and 'jumps'



Observe:
Clock condition holds always,
Its converse does not hold always

Total Order

- <u>Ordering Events Totally</u>

  It is often useful to be able to put a total order on the set of all events. We simply order events by the logical time assigned to them. To break ties ($C_i(a) = C_j(b)$), we impose some fixed rule. One such rule could be based on process numbers (assume each process has a unique number).

- We define a relation '$\Rightarrow$' (total order) as follows:

  $a \Rightarrow b$ (read a ordered before b) if and only if $C_i(a) < C_j(b)$,

  or,

  $C_i(a) = C_j(b)$ and $P_i < P_j$.

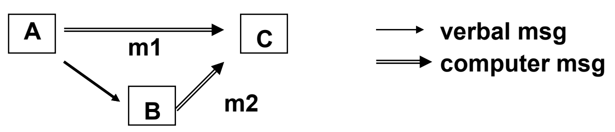
  Notes: (i) if $a \rightarrow b$ then $a \Rightarrow b$;
  (ii) while $\rightarrow$ is unique for a given set of events, there could be several $\Rightarrow$ orderings, depending upon the rule used to break the ties, and upon the way individual process implements IR1.
  (iii) total ordering based on logical clocks cannot be guaranteed to respect the temporal ordering of events that are not related by $\rightarrow$; this can give rise to anomalous behaviour.

Anomalous Behaviour

T11

- Anomalous behaviour (AB):

  A system S using logical clocks for total ordering can permit anomalous behaviour, whereby events are observed to happen in an order that is not consistent with the ordering indicated by the clock numbers.
- Example:
  - Suppose you send a message m1 (event a) from computer A to computer C and then verbally instruct your colleague on computer B who then sends a message m2 (event b) to computer C. Clearly, b has happened after a, but the timestamp of m2 cannot be guaranteed to be greater than that of m1.



The reason is that in S, a and b are not related by $\rightarrow$, as verbal messages are not part of S.
How can anomalous behaviour be prevented?

Understanding AB

T12

- Let S be the new system including all events of interests (those of S and sending/receiving of verbal messages)
- Let ➔ be the happened before relationship in the new S
- We require the following strong clock condition to prevent anomalous behaviour in (the old) system S:

  For any events a, b in S: if a ➔ b then C(a) < C(b)

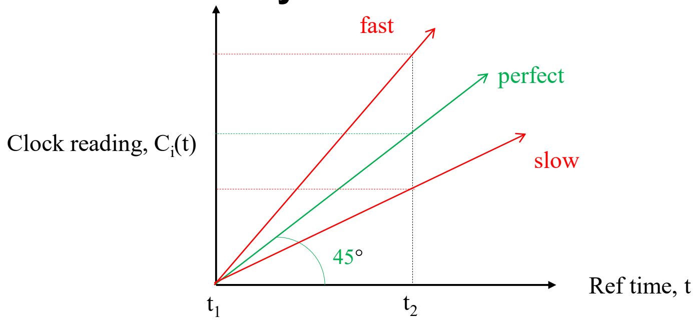
- The strong clock condition cannot be met by logical clocks (as they 'tick' only for events in S)
- However, if clocks can be made to represent physical or standard reference time, then the strong clock condition can be met (mostly)
- So let use introduce physical clocks for time-stamping messages
- Modelling Physical Clocks:

  Let $C_i(t)$ denote the reading of the clock C_i at physical/reference time t.

  Assume that clocks run continuously, rather than in discrete 'ticks'.

Physical Clocks

T13



- R_i: the rate at which C_i runs relative to ref time
  - $(C_i(t_2) - C_i(t_1))/(t_2 – t_1)$
- Ideally, R_i must be equal to 1; if so,
  - $(C_i(t_2) - C_i(t_1)) = (t_2 – t_1)$ and C_i is called a perfect clock
- Perfect clocks hardly exist due to impurities in crystal
- A given physical clock runs either
  - faster: $(C_i(t_2) - C_i(t_1)) > (t_2 – t_1)$ or $R_i > 1$, or
  - slower: $(C_i(t_2) - C_i(t_1)) < (t_2 – t_1)$ or $R_i < 1$

Good clocks and Synch error

T14

A physical clock is said to be good if it does not run too fast or too slow

PC1: there is a known constant k (k << 1), such that for all t:

$1-k \leq$ or $R_i \leq 1+k$ (for a crystal clock, $k \approx 10^{-6}$)

That is, a good clock C_i will measure a time interval d as:

$$\underset{\substack{\uparrow \\ \text{slowest (good) clock}}}{d - kd} \leq C_i(t+d) - C_i(t) \leq \underset{\substack{\uparrow \\ \text{fastest (good) clock}}}{d + kd}$$

It is not enough for the clocks to be good!

They must be synchronised such that the absolute difference between clock readings at any ref time is always bounded by a known constant.

We state this as another condition PC2. Let ε, denoting synchronisation error, be a small known constant such that:

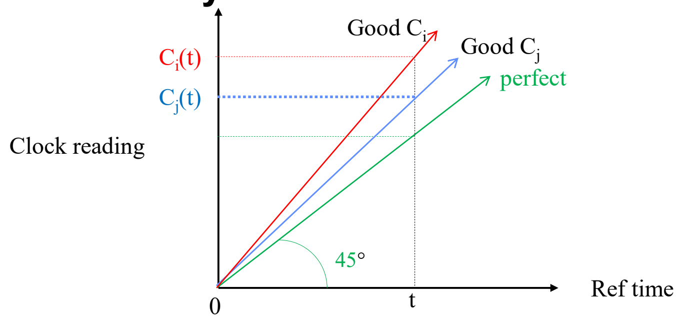
PC2: for all i and j, and for all t :

Either $0 \leq (C_i(t) - C_j(t) \leq \varepsilon$ (assuming C_i is faster than C_j),
or $0 \leq (C_j(t) - C_i(t) \leq \varepsilon$ (assuming C_j is faster than C_i),

Stated differently, PC2: $0 \leq |C_i(t) - C_j(t)| \leq \varepsilon$ where $|C_i(t) - C_j(t)|$ is the difference ignoring the sign

Synchronisation Error

T15



- The above Figure assumes that C_i is faster than C_j (note: both are fast *relative to* the perfect clock)
- When C_i and C_j are synchronised, $|C_i(t) - C_j(t)| \leq \varepsilon$ holds for all t
- Synchronisation is perfect if $\varepsilon = 0$, which is not possible:
  - All clocks need not have identical running rate R, and
- When synchronised clocks are used for time-stamping, m2 (in T11) will certainly have a larger timestamp than m1, if $\varepsilon = 0$
- How small ε should be to prevent anomalous behaviour?

Required smallness of ε

T16

Let μ be the minimum message transmission time in S (see T11)

Note: μ could be the physical distance divided by the speed of light; in practice, it could be significantly larger.

a ➔ b, in T11; if a happens at time t, then b can happen just after t + μ

- To avoid anomalous behaviour, we require: $C_B(t+\mu) - C_A(t) > 0$

  From PC2: the reading of C_B at t, $C_B(t)$, is within ε of the reading of C_A at the same t, $C_A(t)$; so,

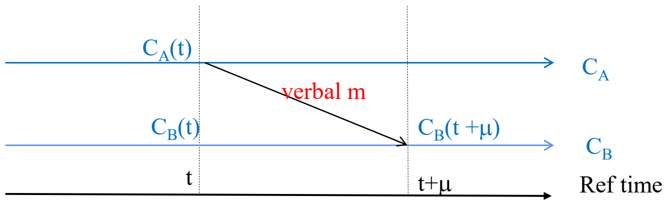
  $C_B(t)$ is a value between $C_A(t) - \varepsilon$ and $C_A(t) + \varepsilon$: $C_A(t) - \varepsilon \leq C_B(t) \leq C_A(t) + \varepsilon$

  Example: When ε =1 min and A's clock = 10:30, B's clock is between 10:29 and 10:31

- Requirement of interest: $C_B(t+\mu) - C_A(t) > 0$ or $C_B(t+\mu) > C_A(t)$
- So, let us consider the smallest value that $C_B(t)$ can take:
  - $C_B(t) = C_A(t) - \varepsilon$
  - the smallest reading for B's clock in the example is 10:29

Required smallness of ε - Contd

T17



- Relating $C_B(t+\mu)$ and $C_B(t)$ using PC1:

  $C_B(t) + (1-k)\mu \le C_B(t+\mu) \le C_B(t) + (1+k)\mu$

  Example: Say $\mu = 10$ and $k = 0.05$. B's clock advances by, in 10 units of ref time: $10(0.95) = 9.5$ if C_B is the *slowest* good clock,

  $10(1.05) = 10.5$ if C_B is the *fastest* good clock,

  As before, let us consider the smallest possible value for $C_B(t+\mu)$:

  $C_B(t+\mu) = C_B(t) + (1-k)\mu$

  Using the previous result ($C_B(t) = C_A(t) - \varepsilon$), we relate $C_B(t+\mu)$ and $C_A(t)$:

  $C_B(t+\mu) = C_A(t) - \varepsilon + (1-k)\mu$; $C_B(t+\mu) - C_A(t) = (1-k)\mu - \varepsilon$

  To avoid AB, we require $C_B(t+\mu) - C_A(t) > 0$; so, $\varepsilon < (1-k)\mu$

  Avoidance of anomalous behaviour is guaranteed, if $\varepsilon < (1-k)\mu$

Synchronising Physical Clocks

T18

- We have seen the need to have small ε - synchronisation error
- Physical clocks do drift apart, due to non-identical run rates
- So, Periodically their readings must be adjusted to keep ε small

Two Problems to be solved:

(i) how to adjust the clock reading

(ii) by how much

How To:

Hardware Approach:

- adjust the running rate directly while crystal oscillations are being turned into clock 'tick's

Software Approach:

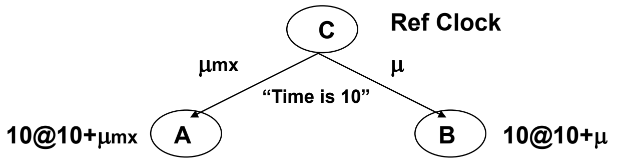
- physical clock is a read-only register
- hence construct an abstraction: synch_clock = physical_clock + Adj, where Adj is a register containing the necessary adjustment at any given time.
- updating Adj should not be discrete, but be gradual over a period of time.
  - jumps or rollbacks in clock readings should be avoided

Computing Clock Adjustments

- Periodically, clocks adjust their readings by some ‘right’ amount. We will see three approaches to obtain this adjustment amount

Centralised Approach:

- A reference clock periodically transmits its value
- it may be one designated clock in the system or an external reference clock (radio time receivers)



- upon receiving the timestamped message, A and B set their readings to the indicated value

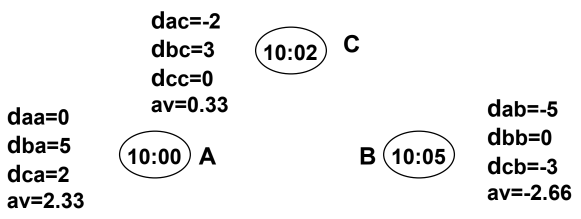
μ_{mx} is the maximum time elapsed between timestamping and receiving

so, the initial error can be (μ_{mx} - μ) which is the maximum variabilities in delays for message processing, queueing and transmission; $\varepsilon > (\mu_{mx} - \mu)$.

Distributed Approach

In Distributed approach,

- each clock reads every other clock
- computes relative difference of every clock with itself
- use the average of these differences as the adjustment amount. Consider a three-clock system and zero message commn delay



- a remote clock has to be read only by message exchange;
  so, $\varepsilon > (\mu_{mx} - \mu)$.
- to avoid network congestion, each clock computes the differences at different (skewed) times, e.g: A reads B and C during [9:45 - 9:50];
  B reads C and A during [9:50 - 9:55];
  C reads A and B during [9:55 - 10:00];

Distributed Approach Contd.

T21

Hardware Approach:

- Use special hardware circuits to
  - (i) timestamp the clock-synch messages just before their transmission onto the physical medium (at the link layer level.)
  - (ii) note the arrival time and copy the timestamp of an incoming clock-synch message soon after it is received from the medium.
- (μ_{mx} - μ) is now only the maximum variabilities in message transmission delays
- can achieve ε in micro seconds

Use of Total Order

T22

- Using total order to construct distributed algorithms
  - processes cooperate by exchanging messages; there is no central synchronising process or shared storage.
- Example: exclusive use of a single resource (mutual exclusion)
  - consider a system composed of a fixed number of processes which share a single resource. Only one process can use the resource at a time, so processes must synchronise themselves to avoid conflicts.
  - An algorithm is required for resource sharing under the following three conditions:

  (i) a process that has been granted the resource must release it before it can be granted to another process;

  (ii) different requests for the resource must be granted in the $\rightarrow$ order in which they were made;

  (iii) if every process that is granted the resource eventually releases it, then every request is eventually granted.

  Note: using a central resource server process for serving requests in the order they were received by the server is not a correct solution (there is no guarantee that the server will receive the messages in increasing timestamp order)

Assumptions and Algorithm

- **Assumptions :**
  - **(i) Events can be ordered according to the total order relation $\Rightarrow$.**
  - **(ii) For any two processes P_i and P_j, messages sent by Pi to Pj are received in the sent order.**
  - **(iii) A process can send a message directly to every other process.**
  - **(iv) Each process maintains a request queue for its own use (the queue is not accessible to other processes)**
  - **(v) Initially: P_0 has the resource and all the request queues have a single message; 'T_0: P_0 requests resource' and T_0 is less than the clock value of any clock of a process.**
- **The algorithm is defined by the following five rules:**
  - **(i) To request a resource, P_i sends the message 'T_m: P_i requests resource' to every other process and also puts this message in its own request queue. (note: T_m is the timestamp of the msg)**
  - **(ii) When a process P_j receives the message 'T_m: P_i requests resource', it places it on its request queue and sends a timestamped acknowledgement message to P_i. (note: the following optimisation is possible: the ack message need not be sent if P_j has already sent a message to P_i with a timestamp greater than T_m.**

Algorithm Rules

(iii) To release the resource, the process using the resource (say P_i) removes any 'T_m: P_i requests resource' message from its request queue and sends 'T_n: P_i releases resource' message to every other process.

(iv) When a process P_j receives a 'T_n: P_i releases resource' message, it removes any 'T_m: P_i requests resource' message from its request queue.

(v) A process (say P_i) is granted the resource when the following two conditions are satisfied:

(a) there is a 'T_m: P_i requests resource' message in its request queue which is ordered before any other request in its queue; and

(b) P_i has received a message with timestamp greater than T_m from every other process.

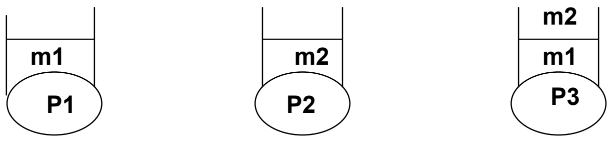
(notes: (i) both the above conditions are tested locally.

(ii) condition (b) above ensures that Pi "knows" that there is no request message that comes before its own request.

Example

T25

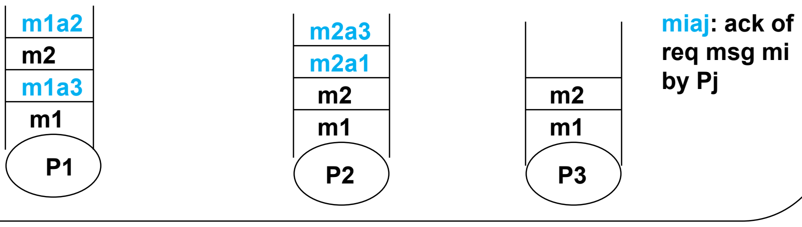
- **Example:**



m1: '6:P1 requests resource', m2: '8:P2 requests resource'

(i) P1 and P2 make requests; m1 gets smaller timestamp than m2; P1 and P2 have not yet seen each other's request.

(ii) P1 (fig. below) can use the resource as its request is ordered before all other requests.



Coursework 1

T26

A Bank has implemented a computerised fire detection system. All the fire detectors within the Bank are connected to a computer (Computer A). Upon receiving a fire signal, computer A will send a "fire!" message to the computer at the local fire station (Computer C), and also to another computer in the Bank (Computer B) that will activate water sprinklers. Computer B is connected to sensors that inform B only if there is NO fire. On being informed by sensors, B deactivates sprinklers and sends a "fire out!" message to computer C. Fire-detectors, sprinklers, sensors and computers within the Bank are adequately fire-proofed and can be regarded to be reliable.

Any message sent is received within a maximum period of *d* time units and *d* is known.

Also known is *e* - the maximum time that can elapse between Computer B activating sprinklers and the sensors informing B if the fire has indeed been put out.

Coursework 1 Continued

T27

The fire station will send the fire engines to the Bank only if no “fire out!” message comes within (2*d*+*e*) time after a “fire!” message has been received.

Figure (a) shows a case which does not require fire engines to be sent.

Figures (b) and (c) show cases where fire engines need to be sent.

In Figure (b), two fire-detectors signal A and the fire is not put out by the water sprinklers,

In Figure (c) a second fire starts after the first one has been put out and it cannot be put out by the water sprinklers and the Bank is burnt down.

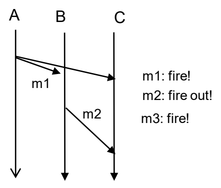


Fig. (a)

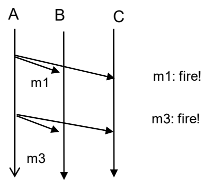


Fig. (b)

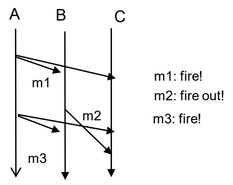


Fig. (c)

Coursework 1 - Questions

T28

I. (i) The system design is flawed. So, it is required that a correct system design be developed in which

1. C only receives messages from A and B, and does not send any to A or B;
2. B only receives messages from A and does not send any message to A;
3. A and B implement logical clocks as per IR1 and IR2 but the only events that advance logical clocks are *send* and/or *receive* events; and,
4. The approach is to choose an appropriate increment for each logical clock

State and explain the requirement(s) that the time-stamps received by *fire* and *fire-out* messages must satisfy so that if sending of a Fire message (e.g., m3) and sending of a Fire-out (m2 in Fig c) are concurrent, they can be distinguished and a new time-out is set to decide whether dispatching of fire engines is essential.

You may use figures for clarity in explanations. Propose some logical clock increments for A and B that meet the requirement and show through an example that the design is correct.

(5 marks, 4 pages max)

II. Suppose that A and B use physical clocks (instead of logical clocks) for time-stamping their respective messages, and that the clocks are synchronized within a known bound ε. Under what conditions, the requirements of (i) would be met, if at all. (3 Marks, 2 pages max)

Submission & Mark Scheme

T29

- **Submission Format**:
  - Word or PDF,
  - Pages refer to A4 size, 1- or 2- columned
  - Page limit includes figures, appendices and any references.
  - Submission in NESS by the prescribed deadline
  - You do not have to refer to transparencies in lecture handouts
- **General Mark Scheme** (for each sub-question):
  - 50% for correctness and 50% for clarity of write-up given that solutions are correct
  - Totally incorrect answers receive no marks however well presented
- Notes:
  - There is no one correct answer, even though there is certainly one most common and correct answer.
  - Your answer will be judged based only on its own merit – that is, how correct it is and how well it is presented
  - It will NOT be judged based on how close it is to the common correct answer
  - Each submission will be marked by the module leader, not by demonstrators.

Learning and Skills Outcomes

T30

- **Learning Outcomes:**
  - **Deeper understanding of 'happened before' relation in a distributed system**
  - **Importance of logical clocks and their use**
  - **Significance of clock error in synchronized physical clocks**
  - **Merits/shortcomings of logical/physical clocks**
  - **Foundations for Coursework 2**
- **Skills Outcomes:**
  - **Analytical**
  - **Presentational (writing)**

Coursework 2: Designing a Total Order (TO) service

CONTEXT:

- Consider a distributed system of *n*, *n* > 3, processes: P1, P2, …, P*n*
- A process can generate an (application-related) message *m* and send *m* to an arbitrary subset of processes in the system
  - There is no assumption that *m* must only be sent to every other process in the system
- Each process has an instance of TO service implementation in its local node.
  - Let us call the instance of TO for process Pi , 1 ≤ i ≤ *n*, TO_i.
- When, say, P1 sends *m* to P2 and P3, it gives *m* to local TO_1 which ***immediately*** timestamps *m* with a logical cock, increments the clock (IR1 of #T7) and sends *m* only to the indicated destinations.
- When *m* reaches the host node of, say, P2, the TO_2
  - Receives *m* (on P2's behalf) while implementing IR2 and IR1 before and after receiving *m* respectively,
  - Puts *m* in a local queue as per the timestamp, and
  - Deduces the right moment to hand-over *m* to P2
- TO_i handing over a received *m* to local Pi is called delivering *m*
- Delivering *m* is an irreversible operation;
- TO_i cannot deliver *m* to Pi, change its mind, un-deliver *m* and then re-deliver *m*.

Coursework 2: Specification

- The Total Order service must satisfy the following two conditions in delivering the received messages to processes:

  (C1) Say messages m and m' are sent to process Pi. If sending of m *happened before* sending of m', then the delivery of m to Pi (by TO_i) must happen before the delivery of m' to p.

  (respects "happened before")

  (C2) Say messages m and m' have common destinations of, say, Pi and Pj. The deliveries of m and m' to Pi and Pj (by TO_i and TO_j respectively) must occur in an identical order:

  either the delivery of m precedes the delivery of m' at both p and q
  or delivery of m' precedes delivery of m at both p and q

- Service should NOT
  - Require a TO_i to receive and then either discard or forward a message not destined for itself or Pi
  - Every message or ack that TO_i receives must carry some useful info for implementing the service

Assumptions

T33

- (A1) There is NO access to synchronised physical clocks;
- (A2) A message by a source TO is eventually received at a destination TO, but no known bound on transmission delays;
- (A3) Between any two TO_i and TO_j, messages sent by TO_i to TO_j are received in the sent order;
- (A4) Sending of a message to one or more destinations may be considered as a single event at the sending end (so, copies of a message sent to multiple destinations have the same timestamp),
- (A5) Processes are uniquely ordered; this ordering is known to all TO processes, and
- (A6) Processes do not crash

Coursework 2 : Requirement

T34

Required: A distributed design for the total order service.

(12 Marks, 6 Pages max)

- You **must describe your design** in terms of how a TO_i carries out its
  - Message sending task
  - Message receiving task, and
  - Message delivery task
- You may assume, if required, that the sending task of TO_i periodically sends a time-stamped 'hello' message to every TO_j, j ≠ i.
- You **must also provide** a description with examples explaining how your design meets its specification, i.e., C1 and C2.
  - To keep within the page limit, you are advised to think and select such examples that explain multiple cases/aspects
- Correctness of design and clarity of explanation are the only criteria for marking this coursework
  - This **trivial design** meets C1 and C2: do not deliver any received message; delivery task simply discards every received m; this design **will receive zero** marks
- Submission and Mark scheme criteria are as in Coursework 1

Coursework 2 : Hints

T35

A TO_i has a queue of received messages that are due for local P_i.

The Transparency T10 ensures that the messages, when ordered as per the logical clock timestamps, enforce a total order – captured in conditions C1 and C2 of the specification

The problem for TO_i is this:

Say, m with C(m)= 5 is at the head of the queue. It has to know what it should observe to conclude that it can NEVER receive another m' with C(m') ≤ 5? Because, if TO_i delivers m to Pi (for processing) and then receives m' with C(m') = 4 arriving, it would have failed Pi. Why?

Both m and m' are destined for Pi, C(m') < C(m), and m' is before m in the total order; but m is delivered before m'!

So, the design challenge here is: *What should each TO_i should do, in addition to (i) implementing the local logical clock (ii) sending and receiving messages on behalf of P_i (iii) and queuing the received messages?*

Learning and Skills Outcomes

T36

- Learning Outcomes:
  - deeper understanding of design issues in accomplishing total order on messages
  - Relating mutual exclusion as a total order issue in distributed systems
- Skills Outcomes:
  - Analytical
  - Application of facts for system design
  - Use of examples to demonstrate correctness
  - Presentational (writing)