Event ordering in a Distributed System

SHUBHABRATA SEN

Outline

- Introduction to distributed computing
- Event ordering in distributed systems
- The ‘happened-before’ relation
- The ‘happened-before’ relation with logical clocks
- Ordering events using the clock conditions
- The event ordering algorithm
- Limitations
- Avoiding anomalous behaviour
- Ordering events using physical clocks
- Conclusion

Distributed Computing

- **Distributed computing** – a field of computer science that studies distributed systems. A distributed system is a software system in which components located on networked computers communicate and coordinate their actions by passing messages
- **Characteristics of distributed systems** – concurrency of components, lack of a global clock, independent failure of components, structure of the system is not known apriori, and limited system view of each system component
- **Examples and applications**
  - Telephone/Cellular Networks
  - Internet
  - P2P networks
  - Massively Multiplayer Online games
  - Distributed databases

Event ordering in distributed systems

- Leslie Lamport - Time, clocks, and the ordering of events in a distributed system (1978).
- **Distributed computing model** – multiple network components interacting and communicating with each other via message passing
- Distributed systems are asynchronous by nature
- **Key challenge** – how to order or sequence the events in a distributed system to ensure that the final outcome is correct/consistent
- **Example** – sequencing booking requests in a flight reservation system
- Time based sequencing of events fundamental to human nature
- Notion of time related to a personal perception
- Does the same assumption hold in a distributed system ?

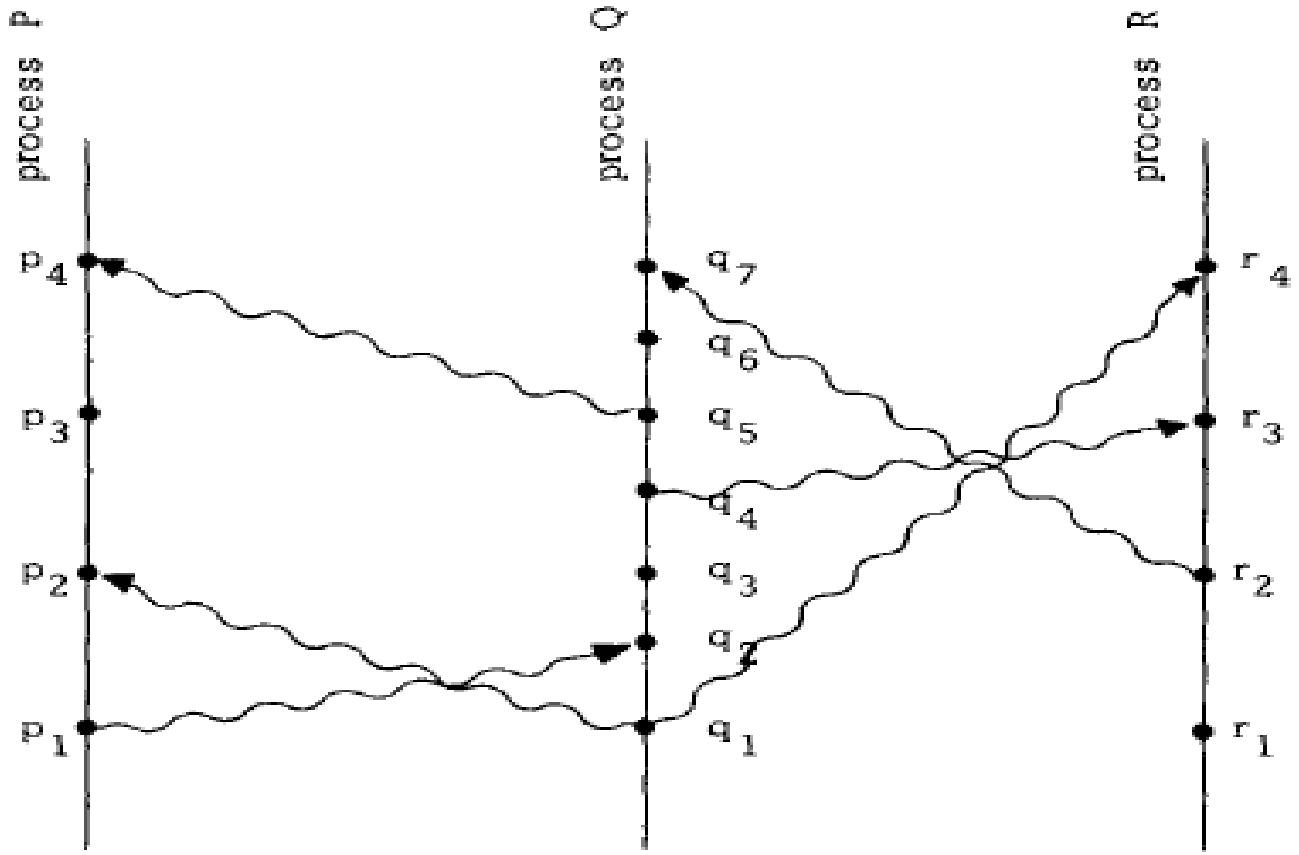
Event ordering in distributed systems

- Reliance on multiple clocks of different system components to observe time can lead to contention issues
- Standardization of the notion of ‘time’ and which event ‘happens-before’ another event is an essential design requirement in a distributed system
- Standard notion of the ‘happened-before’ relation – event A happened before event B implies event A occurs at an *earlier time* than event B
- Justification is based on the physical theories of time
- Defining the ‘happened-before’ relation without using physical clocks

The ‘happened-before’ relation

- **System description** – A collection of processes with each process comprising of a sequence of events
- **Events** – execution of a subprogram, receiving and sending messages
- Events of a process form a sequence where event a *occurs* before event b if event a *happens before* event b
- **Defining the “happened before” relation** “→”
  - If a and b are events in the same process and a comes before b, then a → b
  - If a denotes a message sent by one process and b denotes the receipt of the same message by another process, then a → b
  - If a → b and b → c, then a → c
- “→” represents a partial ordering on the set of events in the system and also reflects the causal relationship between events

Visualizing the ‘happened-before’ relation



REFERENCE - LAMPORT, L. (1978). TIME, CLOCKS, AND THE ORDERING OF EVENTS IN A DISTRIBUTED SYSTEM. COMMUNICATIONS OF THE ACM, 21(7), 558-565

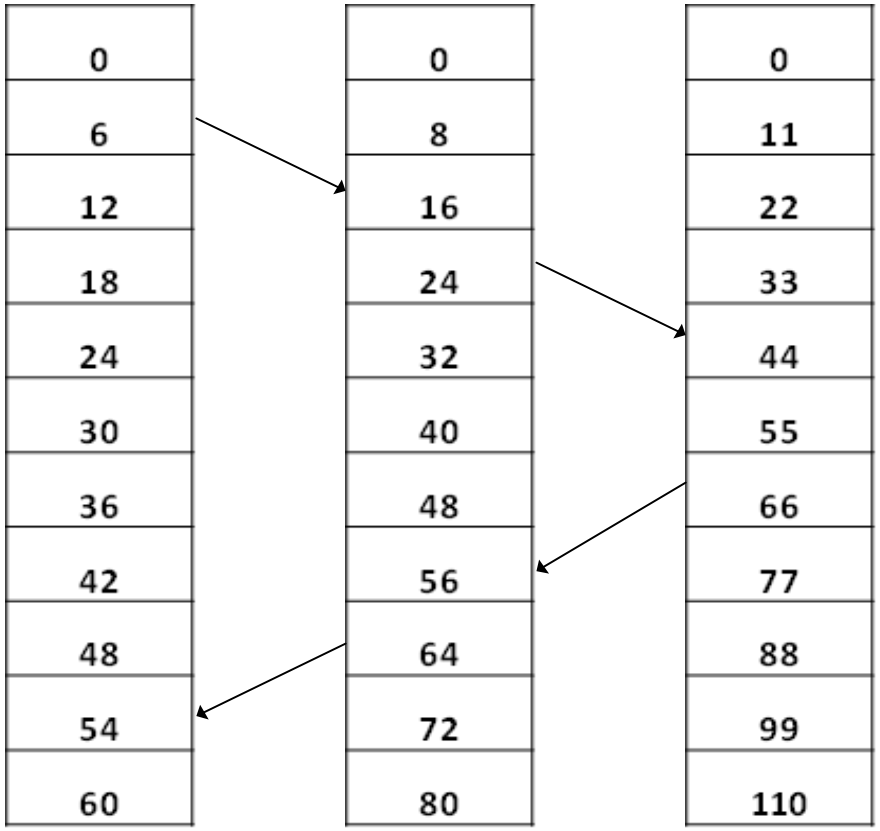
The ‘happened-before’ relation with logical clocks

- **Logical clocks** – abstract way of assigning a number to an event where the number denotes the time of occurrence of the event
- A clock C_i is defined for a process P_i which assigns a number $C_i(a)$ to any event a in that process
- The function C represents the entire system of clocks in the system and assigns numbers to events within the different processes
- Logical clocks implemented using counters with no relation to the physical time
- Evaluating the correctness of a system based on logical clocks
- Correctness definition based on the order in which events occur

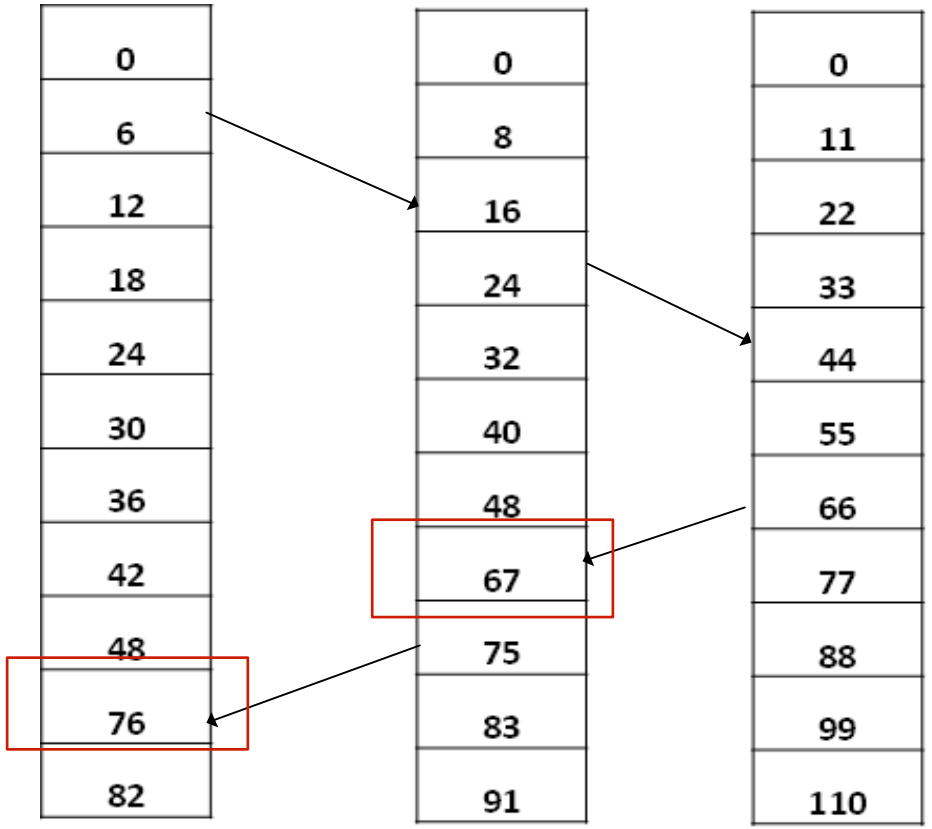
The ‘happened-before’ with logical clocks

- **Clock Condition** – For any events a, b if a → b then C(a) < C(b)
- **Satisfying the Clock condition**
  - C1. If a and b are events in process P_i and a comes before b, then $C_i(a) < C_i(b)$
  - C2. If a is the sending of a message by process P_i and b is the reception of that message by process P_j, then $C_i(a) < C_j(b)$
- **Implementation rules to ensure that a system of clocks satisfy the clock condition**
  - IR1 – Each process P_i increments its logical clock C_i between the occurrence of successive events
  - To satisfy C2, each message m should contain a timestamp T_m that indicates the time when the message was sent
  - IR2 (a) – If an event a in process P_i sends a message m, then $T_m = C_i(a)$
  - IR2 (b) – When process P_j receives a message, it sets the value of its clock C_j to greater than or equal to its present value and greater than T_m

Logical clocks visualization



Processes with independent clocks



Corrected clocks using Lamports algorithm

Ordering events using the clock conditions

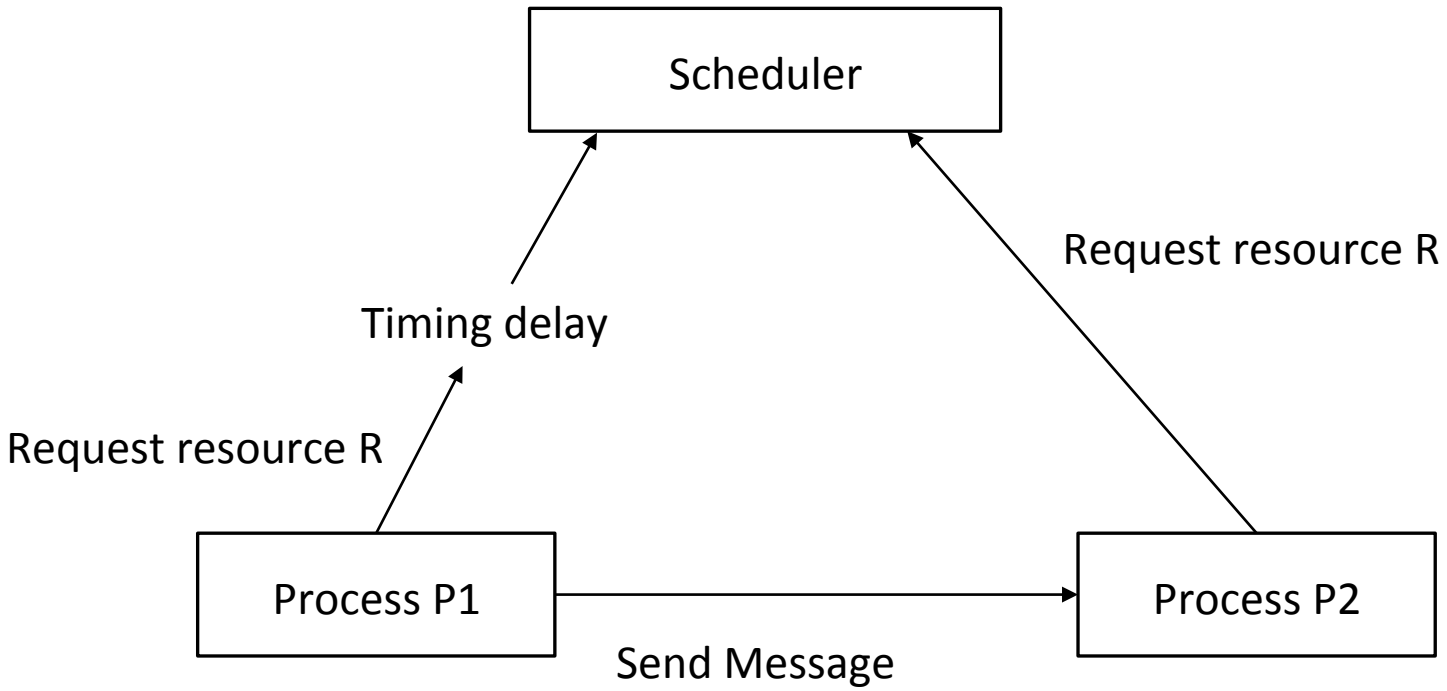
- Use the concept of logical clocks to order the events according to the time they occur
- In case there is a tie between the clock times of two processes in the ordering, use an arbitrary ordering "<" (such as process priority) to break the ties
- Define a new "happened-before" relation "=>" with the following rules - If a is an event in process P_i and b is an event in process P_j, then a => b if and only if either of the two conditions hold
- $C_i(a) < C_j(b)$
- $C_i(a) = C_j(b)$ and $P_i < P_j$
- "=>" completes the "happened-before" partial order relation to a total order relation
- The "=>" ordering is dependent on the system of clocks

Ordering events using the clock conditions

- Usefulness of the event ordering algorithm – solving the mutual exclusion problem
- Synchronizing the access of a single shared resource among multiple processes
- The algorithm should satisfy the following rules
- A resource granted to a process must be released before granting it to another process
- For a given resource, requests must be granted in the order in which they were made
- Assuming each process releases the resource granted to it at some point, all requests to that resource can be fulfilled

Ordering events using the clock conditions

- **Non-trivial problem** – A centralized scheduler granting requests in the order in which they are received will not always work correctly



The event ordering algorithm

- Each process maintains a request queue initially containing a message of the form T_0:P_0
- P_0 – the process initially holding the resource
- T_0 – a value less than the initial values of all the other process clocks
- The algorithm comprises of a collection of rules that govern the synchronization between different processes while accessing a single shared resource
- **Resource request rules**
- The requesting process P_i sends a message T_m:P_i to all the other processes and adds this message to its request queue
- When a process P_j receives this message, it adds it to its own request queue and sends a timestamped acknowledgement to P_i

The event ordering algorithm

- **Resource Release rules**
- Process P_i removes message T_m:P_i from its request queue and sends a timestamped *release resource message* to the other processes
- When a process P_j receives a release resource message, it removes the corresponding T_m:P_i request resource message from its request queue
- **Resource granting rules for process P_i**
- There is a request message T_m:P_i in its request queue and it is ordered before any other request in the queue using the total order 'happened-before' relation "=>"
- P_i has received a message from every other process timestamped later than T_m
- The ordering algorithm is distributed and does not require a centralized scheduler
- Can be extended to define a suitable synchronization behaviour for distributed systems

Limitations

- Requires the active participation of all the processes for successful completion
- All message exchanges between the processes must occur without failures
- If a single process stops working, synchronization problems will occur
- The use of logical clocks may not be feasible in real world scenarios and result in anomalous behaviours
- Person A issues request A on Computer B
- Person A calls friend in another city to issue another request B on Computer B
- Request B receives lower timestamp and is ordered before A – Violation of “happened-before” relation

Avoiding anomalous behaviour

- Define a set of events “L” that comprise of the system events as well as relevant external events
- “happened-before” relation for set L is defined by “→”
- **Avoiding anomalies**
  - Explicitly introduce the additional information about the ordering →
  - Construct system of clocks satisfying the strong clock condition
- **Strong clock condition** – For any events a, b in L : if a → b then C(a) < C(b)
- The strong clock condition is not generally satisfied using the system of logical clocks
- Physical clocks need to be used to eliminate anomalous behaviour

Event ordering with physical Clocks

- Introduction of physical clocks into the existing system setup – $C_i(t)$ denotes the reading of clock C_i at physical time t
- Assume clocks run continuously, then $dC_i(t)/dt$ represents the rate at which the clock runs at time t
- For a true physical clock, $dC_i(t)/dt \approx 1$ for all t
- **PC1**: There exists a constant $\kappa << 1$ such that for all i : $| dC_i(t)/dt - 1| < \kappa$
- $\kappa \leq 10^{-6}$ for crystal controlled clocks
- All clocks must be synchronized so that $C_i(t) \approx C_j(t)$ for all i, j and t
- **PC2:** There exists a sufficiently small constant ε such that for all i,j: $| C_i(t) - C_j(t)| < \varepsilon$

Event ordering with physical Clocks

- Let μ be less than the shortest transmission time for inter process messages
- To avoid anomalous behavior: $C_i(t + \mu) - C_j(t) > 0$
- Based on Condition 1: $C_i(t + \mu) - C_i(t) > (1-k)\,\mu$
- Using Condition 2 it can be shown that: $C_i(t + \mu) - C_j(t) > 0$ if $\varepsilon \leq (1 - \kappa)\mu$
- Implementation rules to ensure that condition 2 holds
- Let m be a message sent at physical time t and received at time t'. Define $v_m = t' - t$ as the total message delay
- Assume receiving process knows a minimum delay μ_m such that $\mu_m \leq v_m$
- $\xi_m = v_m - \mu_m$ defined as the unpredictable delay of the message

Event ordering with physical Clocks

- **IR1'** – For each i, if process P_i does not receive a message at time t, then C_i is differentiable at t and $dC_i(t)/dt > 0$
- **IR2'** - On receiving a message m at time t', process P_j sets C_j (t') equal to $Max(C_j(t'), T_m + \mu_m)$
- Satisfying the physical clock condition PC2
- System of process described by a directed graph where an edge from P_i to P_j represents a communication line
- A message is sent over this edge every τ seconds if for any t, P_i sends at least one message to P_j between physical times t and $t + \tau$
- Diameter of the graph d is the smallest number d such that for a pair of processes (P_j, P_k), there is a path from P_j to P_k having at most d edges

Event ordering with physical Clocks

- **Theorem –** Given a directed graph with diameter d that obeys rules IR1’ and IR2’, PC2 is satisfied with $\epsilon \approx d\,(2k\,\tau + \xi)$
- Proof of the theorem is beyond the scope of this discussion
- A system of physical clocks that are synchronized using the preceding set of rules and conditions can be used to order the events in a distributed system

Conclusion

- Solving the problem of synchronizing the use of a shared resource among events in different processes in a distributed system is a non-trivial problem
- The use of the “happened-before” relation to establish an ordering among the different events
- An invariant partial ordering can be established amongst the events using the concept of logical clocks
- The partial ordering can be extended to a total ordering to solve the synchronization problem
- Anomalous behaviour can occur as a result of using logical clocks
- In order to prevent the anomalies, properly synchronized physical clocks can be used