Distributed Computing Q&A

Past Question Paper

Q1. Explain in detail the role of Middleware services in distributed computing application and the relation between the software and hardware components of machines participating in distributed computing system. [5M]

Middleware in distributed computing acts as the “glue” that enables communication, coordination, and interoperability between heterogeneous hardware and software systems. It provides a uniform abstraction so that applications can run seamlessly across different machines without being aware of the underlying differences in operating systems, network protocols, or hardware architectures. Typical middleware services include communication (RPC, message passing), naming and directory services, distributed file systems, transaction processing, and security.

The relation between software and hardware in a distributed system is mediated by middleware. At the bottom, hardware provides CPU, memory, storage, and networking interfaces. The operating system sits on top of hardware, managing resources locally. Middleware extends this by creating a layer that masks distribution and heterogeneity, making remote resources appear local. Applications then interact with middleware APIs rather than hardware directly, achieving portability, scalability, and transparency in distributed computing.

Q2. Explain consistent global state and its implementation in terms of Cuts with a suitable example. [5M]

A consistent global state in distributed systems refers to a collection of local states from all processes and the states of communication channels such that it represents a possible execution of the system. Since no global clock exists, global states are inferred through logical reasoning.

The concept of a *Cut* helps to reason about global states. A Cut is a set of events, one from each process timeline, which “cuts across” the distributed execution. A Cut is said to be consistent if it does not contain the effect of an event without containing its cause.

For example, suppose process P1 sends a message m to process P2. If the Cut includes the “receive(m)” event on P2 but excludes the “send(m)” event on P1, then it is an inconsistent Cut. Conversely, if both send and receive are included, or both excluded, the Cut is consistent. Consistent Cuts are essential in checkpointing and recovery, where the system needs to capture a valid global snapshot.

Q3. Explain Termination detection using weight throwing with suitable example. [5M]

Termination detection is required in distributed algorithms to decide when all processes have become passive and no messages are in transit. The *weight throwing* technique is a credit distribution method to detect termination.

The initiator starts with a weight of 1 and distributes fractions of this weight along with computation messages to other processes. When a process completes its work and becomes passive, it returns its accumulated weight to the sender. The initiator collects returned weights, and when the total weight equals 1 again and the initiator is passive, termination is declared.

Example:

Suppose P1 initiates computation with weight 1. It sends half (0.5) to P2 with a task, keeping 0.5. P2, after finishing, sends back 0.5. Meanwhile, P1 also completes and holds 0.5. Once P1 regains the full weight of 1 and no messages are in transit, the algorithm detects global termination.

Q4. Compare the Moving sequencer algorithms and Fixed sequencer algorithms. [5M]

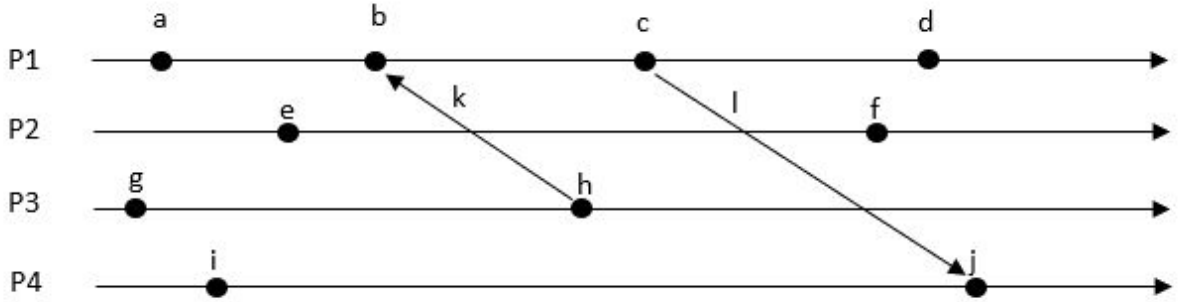
In distributed systems, sequencer algorithms order multicast messages to ensure consistency.

- **Fixed Sequencer Algorithm:**
  A single process is designated as the sequencer. All messages are sent to the sequencer, which assigns sequence numbers and multicasts them to all processes. This ensures total ordering.
  Advantages: Simple and easy to implement.
  Disadvantages: The sequencer can become a bottleneck and a single point of failure.

- **Moving Sequencer Algorithm:**
  The role of sequencer rotates among processes, or the sender itself may act as a temporary sequencer. Sequence numbers are distributed in a round-robin or dynamic manner.
  Advantages*:* Avoids bottlenecks, increases fault tolerance, and balances load.
  Disadvantages*:* More complex coordination is required, and care must be taken to prevent duplicate or conflicting sequence numbers.

Thus, fixed sequencers trade simplicity for reduced scalability, while moving sequencers offer better distribution at the cost of complexity.

Q5. Consider following figure that shows four processes (P1, P2, P3, P4) with events and messages communicating between them.

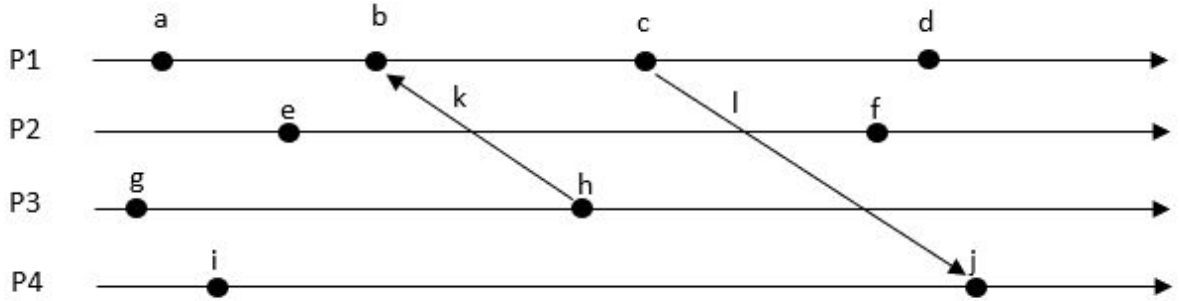


a. Redraw the time space diagram and list the Vector Clock values in the given table for each event and message passed to other processors shown in Figure . Assume that each process maintains a logical clock as a single integer value and d=1. Provide timestamps for each labeled event. [5M]

b. Redraw the time space diagram and list the Fowler–Zwaenepoel's direct-dependency technique values in the given table for each event and message passed to other processors shown in Figure. Assume that each process maintains a logical clock as a single integer value and d=1. Provide timestamps for each labeled event.[5M]

Assumptions for timestamp calculation:

- **Processes:**
  P1 executes events $a \rightarrow b \rightarrow c \rightarrow d$, P2 executes $e \rightarrow f$, P3 executes $g \rightarrow h$, and P4 executes $i \rightarrow j$.
- **Messages:**
  k represents a message sent from **P3** and received at **b (P1)**.
  l represents a message sent from **c (P1)** and received at **j (P4)**.
- **Vector Clocks:**
  Each process maintains a vector clock of four components in the order [P1, P2, P3, P4]. For every local event, the process increments its own component by **d = 1**. On receiving a message, the process updates its vector clock by taking the component-wise maximum of its local clock and the piggybacked clock, then increments its own component.



Vector clock table

a [1,0,0,0]
b [2,0,2,0]
c [3,0,2,0]
d [4,0,2,0]
e [0,1,0,0]
f [0,2,0,0]
g [0,0,1,0]
h [0,0,2,0]
i [0,0,0,1]
j [3,0,2,2]
k [0,0,2,0] <-- message P3→P1 (piggybacked at send)
l [3,0,2,0] <-- message P1→P4 (piggybacked at send)

Fowler–Zwaenepoel direct-dependency values for each label

a {P1}
b {P1,P3}
c {P1,P3}
d {P1,P3}
e {P2}
f {P2}
g {P3}
h {P3}
i {P4}
j {P1,P3,P4}
k {P3} <-- message P3→P1 (piggybacked at send)
l {P1,P3} <-- message P1→P4 (piggybacked at send)

Fowler–Zwaenepoel's direct-dependency method

In Fowler–Zwaenepoel's direct-dependency method, each event is associated with the set of processes on which it directly depends. Initially, every process depends only on itself. For local events, the dependency set does not change — it is simply inherited from the previous event on that process. When a process sends a message, it attaches its current dependency set to that message. The receiver, upon getting the message, forms the union of its own dependency set with the one carried by the message, and then includes itself. This is the only moment when the set can expand.

Applying this to the table: events a, e, g, and i have single-element sets because they are purely local. At b, P1 receives a message from P3, so {P1} merges with {P3} to give {P1, P3}. Events c and d remain {P1, P3} because no further messages arrive. At j, P4 receives a message from P1 that carries {P1, P3}; union with {P4} gives {P1, P3, P4}. Meanwhile, P2 is isolated, so its events remain {P2}. Messages k and l themselves carry the sets {P3} and {P1, P3} respectively.

The outcome is that dependency sets only grow at receive events, and otherwise remain stable. This provides a compact way to express causal relationships without needing full vector clocks.

Assumptions for timestamp calculation:

- **Processes:**
  P1 executes events $a \rightarrow b \rightarrow c \rightarrow d$, P2 executes $e \rightarrow f$, P3 executes $g \rightarrow h$, and P4 executes $i \rightarrow j$.
- **Messages:**
  k represents a message sent from **P3** and received at **b (P1)**.
  l represents a message sent from **c (P1)** and received at **j (P4)**.
- **Vector Clocks:**
  Each process maintains a vector clock of four components in the order [P1, P2, P3, P4]. For every local event, the process increments its own component by **d = 1**. On receiving a message, the process updates its vector clock by taking the component-wise maximum of its local clock and the piggybacked clock, then increments its own component.
- **Fowler–Zwaenepoel Direct-Dependency:**
  For this method, each event is associated with the set of processes it directly depends on. A sender piggybacks its current dependency set along with a message. Upon receiving, the receiver forms the union of its own set with the sender's set, ensuring direct dependencies are correctly tracked.


BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
ज्ञानं परमं बलम्