**Unit – 5**

**Checkpointing**

**Checkpointing** is the process of saving the current state of a system or application at regular intervals. It enhances fault tolerance and ensure efficient recovery from system failures. In the event of a system failure, the system can revert to a previously saved state, minimizing data loss and the need for a full restart.

**Types of Checkpointing**:

* **Coordinated Checkpointing**: All processes save their state simultaneously, ensuring consistency across the system.
* **Uncoordinated Checkpointing**: Processes save their states independently, which can lead to issues like inconsistency but has lower overhead.
* I**ncremental Checkpointing**: Only changes since the last checkpoint are saved, reducing storage overhead.
* **Application-level Checkpointing**: The application itself is responsible for saving its state.
* **System-level Checkpointing**: The underlying system automatically checkpoints the entire process or virtual machine.

**Benefits**:

* **Fault Tolerance**: Checkpointing allows applications to recover from faults by restoring the last saved state.
* **Improved System Availability**: Reduces downtime by quickly resuming operations from a known state.
* **Data Integrity**: Prevents data loss by regularly storing the system state.
* **Efficiency in Resource Usage**: Saves computation time and resources in case of system crashes, as re-computation from the start is avoided.

**SLO**

SLO (Service Level Objective) is a measurable goal set for a specific aspect of a service, such as availability, response time, or error rate. It defines what is expected from the system's performance.

Example: An SLO for a cloud storage service could be 99.9% availability, meaning the service should be up and running 99.9% of the time.

SLOs are part of:

* SLAs (Service Level Agreements), where SLOs are formalized between service providers and customers.
* SLIs (Service Level Indicators), which are the metrics used to measure whether the SLO is met.

**Consensus**

**Consensus** in cloud computing refers to the process of achieving agreement among distributed nodes (servers or computing entities) on a single data value or state, even in the presence of failures. Consensus ensures that all nodes in a distributed system have the same view of the data, which is critical for system consistency, reliability, and fault tolerance.

Key Aspects of Consensus in Cloud Computing:

1. **Fault Tolerance**: Consensus mechanisms ensure that a system continues to function correctly even if some nodes fail or behave maliciously.
2. **Coordination**: In a distributed cloud environment, nodes must coordinate with each other to agree on the state of a system, such as data replication or transaction commits.
3. **Consistency**: Consensus is necessary to maintain data consistency across multiple nodes, particularly in cloud databases, file systems, and distributed ledgers.
4. **Decentralization**: Consensus allows cloud systems to operate without relying on a single central authority, making them more robust and scalable.
5. **Communication Delays**: Network latency and partitioning can affect the consensus process, as nodes may not communicate in real-time.

Importance in Cloud Systems:

1. **Data Replication**: Ensures that all replicas of data in a cloud system are consistent and up-to-date.
2. **Transaction Commit**: Ensures that all nodes either commit or abort a transaction, preventing partial updates.
3. **System Reliability**: By using consensus, cloud platforms ensure that failures (e.g., node crashes or message losses) do not lead to inconsistent data or system downtime.

Issues in Forming consensus:

1. **Fault Tolerance**: Some nodes may crash or act maliciously, making it difficult for all nodes to agree on a common decision.
2. **Latency**: Geographic distribution of nodes introduces network delays, making fast consensus difficult.
3. **Asynchrony**: In an asynchronous system, messages can arrive out of order, leading to uncertainty and complications in achieving consensus.
4. **Scalability**: As the number of nodes increases, the complexity of reaching consensus grows, requiring more communication and coordination.
5. **Network Partitions**: Network failures can isolate parts of the system, preventing nodes from communicating and agreeing.
6. **FLP Impossibility**: In asynchronous systems, it’s impossible to guarantee consensus with even a single faulty node without compromising liveness or safety.
7. **Byzantine Failures**: Malicious nodes may send incorrect or misleading information, making it hard to reach reliable consensus.
8. **Leader Election**: In consensus protocols like Paxos, electing a leader who coordinates agreement adds additional overhead and complexity.

**Consensus in Synchronous and Asynchronous Systems**

In distributed computing, consensus protocols are used to achieve agreement on a shared state or value among multiple processes or nodes. The performance and complexity of these consensus protocols depend heavily on whether the system is **synchronous** or **asynchronous**. Here’s a detailed explanation of consensus in both types of systems:

**1. Consensus in Synchronous Systems**

**Synchronous systems** are those in which there is a known, bounded delay for message delivery and a known, bounded processing time for nodes to execute instructions. This means every node in the system has a clear expectation of when messages will be received, and the system behaves predictably with respect to time.

**Characteristics:**

* **Bounded delays**: The system has a guaranteed upper bound on message delivery and processing time.
* **Predictability**: Nodes know the maximum time they must wait for responses from other nodes.
* **Global clock**: Nodes have access to a global clock or equivalent time synchronization, allowing them to coordinate easily.

**Consensus Protocols in Synchronous Systems:**

* **Simplicity**: Achieving consensus in synchronous systems is easier because nodes can rely on a fixed time for message exchanges and decision-making.
* **Examples**: Protocols like **Synchronous Paxos** are designed to work in such systems.

**Advantages:**

* **Deterministic timing**: Faults like message loss or node failures can be detected more quickly, making consensus more efficient.
* **Fault tolerance**: Byzantine and crash faults can be handled more easily in synchronous systems because all nodes can expect to hear from every other node within a known time frame.

**Disadvantages:**

* **Not practical for real-world systems**: Purely synchronous systems are rare in practice because real-world networks often experience unpredictable delays and failures.

**2. Consensus in Asynchronous Systems**

**Asynchronous systems** are those where there are no guarantees on message delivery time, and nodes can experience arbitrary delays in communication or processing. In these systems, there’s no global clock or synchronization, making consensus more complex.

**Characteristics:**

* **Unpredictable delays**: Messages may arrive late or in a different order, and processing times vary.
* **No timing assumptions**: Nodes cannot make assumptions about when they will receive messages from others.
* **Decentralized time**: Each node works independently without relying on a global clock.

**Consensus Protocols in Asynchronous Systems:**

* **FLP Impossibility**: In 1985, the FLP theorem (Fischer, Lynch, and Paterson) proved that achieving deterministic consensus in a purely asynchronous system is impossible if even one node can fail (crash or behave maliciously).
* **Leader-based protocols**: Protocols like **Paxos** and **Raft** work in asynchronous systems by relying on leader election, message retries, and other mechanisms to deal with uncertainty.
* **Byzantine Agreement (BA)**: Consensus in asynchronous systems also includes dealing with **Byzantine failures**, where nodes can behave maliciously or send incorrect information.

**Advantages:**

* **Real-world applicability**: Asynchronous consensus protocols can be applied to real-world distributed systems like the cloud, where delays and faults are unpredictable.

**Disadvantages:**

* **Higher complexity**: Consensus is harder to achieve in asynchronous systems due to the uncertainty in message delays and the potential for node failures.
* **More rounds of communication**: Asynchronous consensus protocols often need multiple rounds of communication to reach agreement, increasing latency.

**Comparison of Consensus in Synchronous vs Asynchronous Systems:**

| **Feature** | **Synchronous Systems** | **Asynchronous Systems** |
| --- | --- | --- |
| **Message Timing** | Known and bounded message delays | No guarantees on message delivery times |
| **Consensus Difficulty** | Easier to achieve due to predictable timing | Harder due to unpredictable delays and failures |
| **Time Synchronization** | Global clock or synchronized time | No global clock, nodes work independently |
| **Fault Tolerance** | Can detect and handle faults faster | Faults may be harder to detect or mitigate |
| **Examples of Protocols** | Synchronous Paxos, simpler Byzantine protocols | Paxos, Raft, Byzantine Agreement in asynchronous settings |
| **Real-World Usage** | Rare due to the need for exact timing | Common in distributed cloud systems |
| **Reliability** | High, as faults are predictable | Requires complex handling of faults and retries |
| **FLP Impossibility** | Not applicable | Proven that consensus is impossible with faults |

**Measuring the Reliability and Performance of Cloud Systems**

1. **Availability**: The percentage of time a service is operational and accessible.
2. **Latency**: The time it takes for a request to travel from the source to the destination and back.
3. **Throughput**: The amount of data processed or transferred over a network within a given time period.
4. **Error Rate**: The percentage of failed or erroneous requests out of the total number of requests.
5. **Response Time**: The time taken by a system to respond to a request.
6. **Uptime**: The amount of time a service remains functional without any interruptions.
7. **Durability**: The ability of the system to retain data over time without loss.
8. **Scalability**: The capacity of a system to handle growth in load by adding resources without compromising performance.

**Byzantine Agreement**

**Byzantine Agreement (BA)** is a consensus mechanism that deals with the challenge of reaching an agreement in the presence of faulty or malicious nodes (Byzantine failures).

* **Byzantine Faults**: These occur when some nodes in the system behave arbitrarily or maliciously, potentially misleading other nodes.
* **Conditions for Byzantine Agreement**:
  1. **Agreement**: All non-faulty nodes must agree on the same value.
  2. **Termination**: Every non-faulty node must eventually decide on some value.
  3. **Validity**: The value agreed upon by all non-faulty nodes must be valid or consistent with the system's rules.
* **Byzantine Generals Problem**: The classic problem that illustrates the difficulty of reaching consensus when some participants (generals) may act maliciously or fail to communicate reliably. It shows that at least **3f+13f + 13f+1** nodes are required to tolerate **fff** Byzantine faults.
* **Byzantine Fault Tolerance (BFT)**:
  1. **Tolerance of Failures**: To tolerate **f** Byzantine nodes, a system needs at least **3f+13f + 13f+1** nodes. This ensures that even if some nodes behave maliciously, the system can still reach consensus.
  2. **Consensus Protocols**: Protocols like **PBFT (Practical Byzantine Fault Tolerance)** are used to handle Byzantine failures in distributed systems. These protocols are designed to ensure that all non-faulty nodes agree on the same value, even in the presence of malicious or faulty nodes.
* **Trade-offs**: Byzantine Agreement protocols tend to be slower and more resource-intensive due to the need for extensive communication to ensure agreement.