Cloud Computing (UE18CS352)
Unit 4

Aronya Baksy

March 2021

1 Master-Slave vs P2P Models

- Distributed System: A system that involves components on different physical machines that communicate, and coordinate actions in order to appear as a single system to the end user
- Two main types of distributed system architectures are **Master-Slave** and **Peer-to-Peer** (P2P)

1.1 Master-Slave Architecture

1.1.1 Key Points

- Two types of nodes: master and worker
- Nodes are unequal, the master is above the worker nodes in the hierarchy. This makes the master a **single point of failure** (SPOF)
- The master is the *central coordinator* of the system. All decisions regarding scheduling and resource allocation are made by master
- The master becomes a performance *bottleneck* as the number of worker nodes increases.

1.1.2 Advantages

- Easy maintenance and security
- Promotes sharing of resources and data between different h/w and s/w platforms
- Integration of services

1.1.3 Disadvantages

- Not scalable as number of workers increase
- The master is a SPOF

1.2 P2P Architecture

1.2.1 Key Points

- No hierarchical relationships between the nodes
- No central coordination, each node takes its own decisions on resource allocation. But synchronization of all the decisions is difficult.
- Theoretically ∞ scalability. No performance bottlenecks exist.
- Peers form groups and offer services/data within group members. Popular data is propagated within the group, unpopular data may die out.
- Peers can form a Virtual Overlay Network on top of the physical topology. Each peer routes traffic through the overlay network.

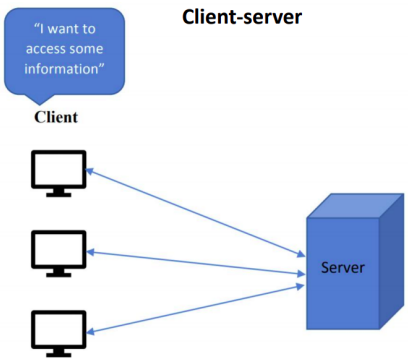

Figure 1: Client-Server Architecture

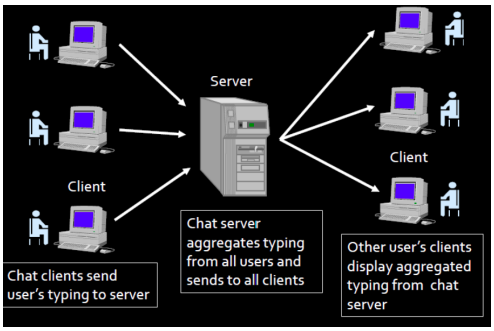

Figure 2: Client Server Model for Chat

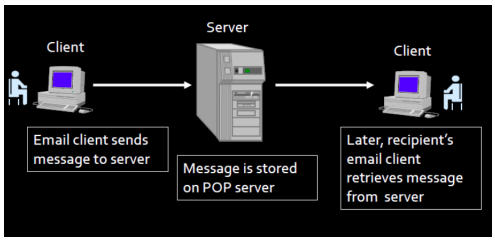

Figure 3: Client-Server Architecture for e-mail applications

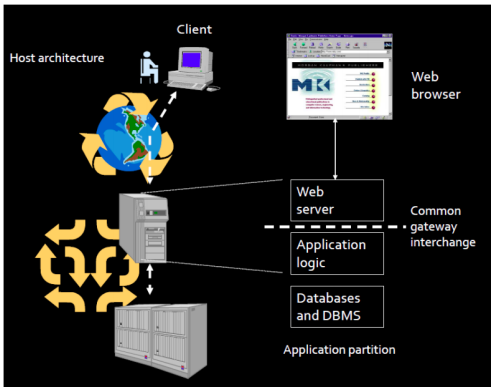

Figure 4: 3-tier Client Server Model

1.2.2 Advantages

- No centralized point of failure.
- Highly scalable, addition of peers does not affect quality of service

1.2.3 Disadvantages

- Maintaining decentralized coordination is tough (consistency of global state, needs distributed coherency protocols)
- Computing power and bandwidth of nodes impacts the performance (i.e. all nodes are not the same in a P2P network)
- Harder to program and build applications for P2P systems due to the decentralized nature.

1.2.4 Applications

- **File Sharing** applications with replication for fault tolerance (e.g.: Napster, BitTorrent clients like μTorrent, Gnutella, KaZaa)
- Large-scale **scientific computing** for data analysis/mining (e.g.: SETI@Home, Folding@Home used for protein dynamics simulations)
- **Collaborative applications** like instant messaging, meetings/teleconferences (e.g.: IRC/ICQ, Google Meet, MS Teams etc.)

1.3 P2P Topologies

1.3.1 Centralized Topology

- A centralized server must exist which is used to manage the files and user databases of multiple peers that log onto it

- The centralized server maintains a mapping between file names and IP addresses of a node. Each time a client joins the network, it publishes its IP address and list of files it shares to this database
- Any file lookup happens via the server. If the file is found then the centralized server establishes a direct connection with the requesting node and the node that contains the requested file.

1.3.2 Ring Topology

- Consists of a cluster of machines that are arranged in the form of a ring to act as a distributed server. The ring provides better load balancing and high availability.
- Typically used when nodes are physically nearby, such as a single organization.

1.3.3 Hierarchical Topology

- Suitable for systems that require a form of governance that involves delegation of rights or authority
- e.g.: DNS hierarchy, where authority flows from the root name servers to the servers of the registered name and so on

1.3.4 Decentralized Topology

- All peers are equal, hence creating a flat, unstructured network topology
- In order to join the network, a peer must first, contact a bootstrapping node (node that is always online), which gives the joining peer the IP address of one or more existing peers.
- Each peer, however, will only have information about its direct neighbours.
- Any file queries have to be flooded to all the nodes in the network.
- e.g.: GNutella, for file sharing especially music

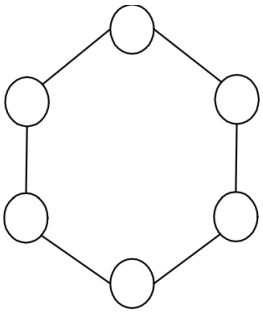

Figure 5: Ring Topology

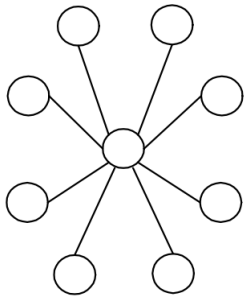

Figure 6: Centralized Topology

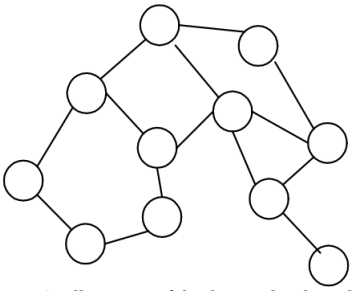

Figure 7: Decentralized Topology

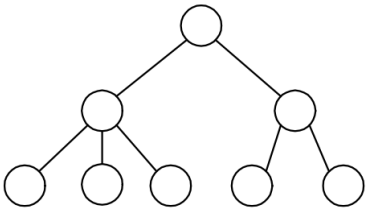

Figure 8: Hierarchical Topology

2 Unreliable Communication

- Issues with communication in distributed systems:
  - Request or response is lost due to issues in the interconnect network
  - Delay in sending request or response (due to queuing delays and network congestion)
  - Remote node failure (permanent or temporary)
- *Partial Failure* in a distributed system occurs when some components (not all) start to function *unpredictably*. Partial failures are *non-deterministic*.
- Distributed systems involve accepting partial failure, building fault-tolerant mechanisms into the system.
- **Reliability** is the ability of a system to continue normal functioning even when components fail or other issues occur.
- Formally, reliability is defined as the probability that a system meets certain performance standards and yields correct outputs over a desired period of time
- Reliability includes:
  - Tolerant to unexpected behaviour and inputs in the software
  - Prevention of unauthorized access and abuse
  - Adequate performance for the given use case under expected load and input size
- Metric for reliability: **mean time between failures**, defined as

$$MTBF = \frac{\text{total uptime}}{\#\text{ of failures}}$$

- A fault is usually defined as one component of the system deviating from its spec
- A failure is when the system as a whole stops providing the required service to the user
- Fault-tolerant mechanisms prevent faults from causing system-wide failures.
- Classification of faults:
  - **Transient**: appear once, then vanish entirely (e.g. first request from node A to node B fails to reach, but the next one reaches on time)
  - **Intermittent**: Occurs once, vanishes, but reappears after a random interval of time. (e.g. loose hardware connections)
  - **Permanent**: Occurs once, interrupts the functioning of the system until it is fixed. (e.g. infinite loops or OOM errors in software)
- Classification of failures:
  - **Crash failure**: A server halts, but functions correctly till that point
  - **Omission Failure**: Could be send omission (server fails to send messages) or receive omission (server fails to respond to incoming messages)
  - **Timing Failure**: server response is delayed beyond the acceptable threshold
  - **Response Failure**: Could be a value failure (response value is wrong for a request) or a state transition failure (deviation from correct control flow)
  - **Arbitrary or Byzantine Failure**: Arbitrary response produced at arbitrary times

2.1 Failure Detection

- Using **timeout**: Let d be the longest possible delivery time (all messages will reach within time d after being sent or they will not reach at all), and r be the time needed by the server to process the message.
- Then the round trip time $2d + r$ is a reasonable estimate for a timeout value beyond which it can be assumed that a node has failed.
- Unfortunately, there are no such time guarantees in asynchronous communications that are used in distributed systems.
- Network congestion causes queuing delays. If queues fill up at routers, then the packets can be dropped, causing retransmission and further congestion
- Even VMs that give up control of the CPU core to another VM can face network delays as they stop listening to the network for that short duration when they are not in control of the CPU. This leads to packets dropping.
- Timeout values are measured *experimentally.*
  - Data is collected on round-trip times across multiple machines in the network and over an extended time period. Measure the variability in the delays (aka **jitter**)
  - Taking into account this data, as well as the application characteristics, a timeout is chosen that is a fair compromise between delay in failure detection and premature timeout.
  - Instead of constant timeouts, the system constantly measures response time and jitter, and dynamically adjusts the timeout value.
  - This is used in Phi Accrual Failure Detector in systems like Cassandra and Akka (toolkit for distributed applications in Java/Scala)
- In circuit switched networks, there is no queuing delay as the connection is already set up end-to-end before message exchange, and the maximum end-to-end latency of the network is fixed (bounded delay)
- The disadvantage of circuit switched network is that it supports far less number of concurrent network users, and it leads to low bandwidth utilization.

2.2 Failure Models

- **Fail-Stop**: Assumes that nodes can fail only by crashing. Once a node stops responding it never responds until it is brought back online
- **Fail-Recovery**: Once a node stops responding it may start responding again after a random time interval. Nodes are assumed to have stable disk storage that persists across failures, but in-memory state is assumed to be lost
- **Byzantine**: A component such as a server can inconsistently appear both failed and functioning to failure-detection systems, showing different symptoms to different observers.

2.3 Byzantine Faults

- A Byzantine fault is a condition of a distributed system where components may fail and there is imperfect information on whether a component has failed.
- A system is said to be Byzantine fault-tolerant if it continues to work properly despite nodes not obeying correct protocol, or if malicious actors are interfering with the working of the system
- The Byzantine agreement:
  - Used to build consensus that nodes have failed or messages are being corrupted
  - General strategy is to have each node communicate not only its own status but any information they have on any other nodes.
  - Using this information, a majority consensus is built and the nodes that don't agree with this consensus are considered to be in failure state.

2.4 Failure Detection using Heartbeats

- A heartbeat is a signal sent from a node to another at a fixed time interval that indicates that the node is alive.
- Absence of a fixed number of consecutive heartbeats from a node is assumed to be evidence that that node is dead
- Heartbeat signals are organized in the following ways:
  - **Centralized**: All nodes send heartbeats to a central monitoring service. Simplest organization but the central service is now a SPOF
  - **Ring**: Each node sends heartbeat only to one neighbour, forming a ring structure. If one of the nodes fails, then the ring breaks and heartbeats cannot be sent properly
  - **All-to-All**: Each node sends heartbeats to every other node in the system. High communication cost but every node keeps track of all other nodes hence high fault tolerance.

2.5 Failover

- The act of switching over from one service/node/application to a new instance of the same, upon the failure or abnormal termination of the first one.
- Failover can be implemented in two ways:

2.5.1 Active-Active Failover Architecture

- Also called symmetric failover.
- Let there be 2 servers. Server 1 runs application A and server 2 runs application B. If server 1 fails for some reason, then server 2 will now be tasked with running both applications A and B.
- Since the databases are replicated, it mimics having only one instance of the application, allowing data to stay in sync.
- This scheme is called Continuous Availability because more servers are waiting to receive client connections to replicate the primary environment if a failover occurs.

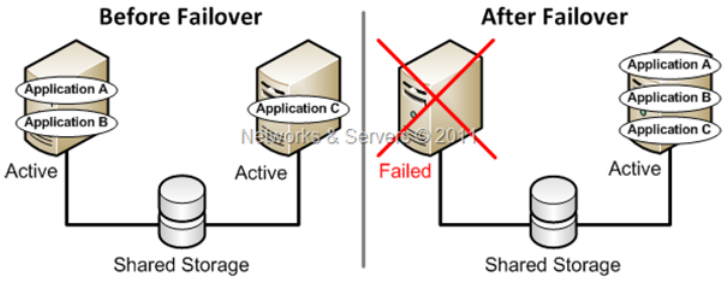

Figure 9: Active-Active Failover Architecture

2.5.2 Active-Passive Failover Architecture

- Also called asymmetric failover.
- A standby server is configured to take over the tasks run by the active primary server, but otherwise the standby does not perform any functions.
- The server runs on the primary node until a failover occurs, then the single primary server is restarted and relocated to the secondary node.
- Not necessary to shift functions back to the primary server once it comes back online (called failback). In this situation the primary is the new standby, and the previous standby is currently the primary.

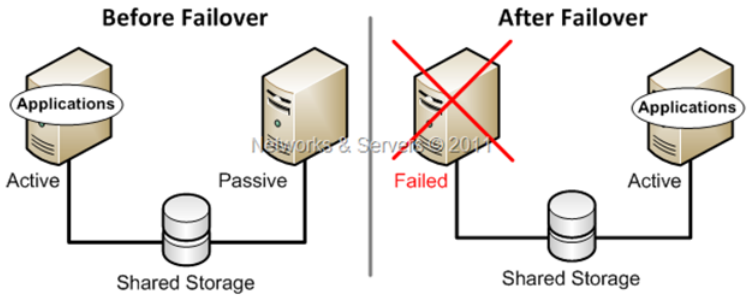

Figure 10: Active-Passive Failover Architecture

3 Availability and Fault Tolerance

3.1 Availability

- Availability is used to describe the situation where a service is ready to respond to user requests, as well as the time spent in actually servicing those requests.
- Uptime refers to the period during which a service is operational. Downtime refers to the period during which a service in unavailable and non-operational.
- Most modern distributed systems give an uptime guarantee of 99.999% (5 nines).
- Downtime may be planned (maintenance is generally planned beforehand at regular intervals) or unplanned (due to network outages, node failures, software crashes)
- The **Service Level Agreement** (SLA) between the cloud provider and end user includes an uptime-downtime ratio.
- Available systems are generally designed by eliminating as many SPOF as possible. As size of system increases, isolation of faults becomes harder hence availability reduces.
- Most distributed systems have **high availability with failover**.

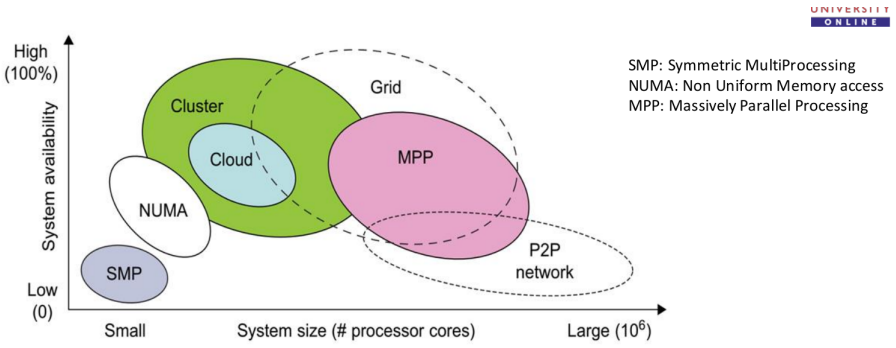

Figure 11: Availability for different types of distributed systems

- Availability is measured in terms of the following metrics:

3.1.1 Mean Time to Failure (MTTF)

- Measured as

$$MTTF = \frac{\text{Total uptime}}{\text{Number of tracked operations/components}} \tag{1}$$

- It is a measure of failure rate of a product
- MTTF is only used for **non repairable, only replacable** components (such as motherboards, memory, disk drives, batteries etc.)

3.1.2 Mean Time to Repair (MTTR)

- Measured as

$$MTTR = \frac{\text{Total downtime caused by failures}}{\text{Number of failures}} \tag{2}$$

- It measures the average time to repair and restore a failed system

In terms of the above metrics, the system availability is defined as:

$$Availability = \frac{MTTF}{MTTF + MTTR} \tag{3}$$

3.2 Fault Tolerance

- Ability of system to continue operation without interruption, despite failure of one or more components.
- Fault tolerance can be implemented either in hardware (duplicate hardware), or in software (duplicate instances running the same software) or using load balancers (redirect traffic away from failed instances).
- Fault tolerant architectures, however, do not address software failures which are the most common cause of downtime in real-life distributed systems.

Consideration	**Highly Available**	**Fault Tolerant**
Downtime	Minimal allowed downtime for service interruption (e.g. 99.999% uptime means 5 mins downtime per year)	Zero downtime, continuous service expected
Cost	Less expensive to implement as no redundant hardware is needed	More expensive to implement as it requires redundant components
Scope	Use shared resources to manage failures and minimize downtime	Use power supply backup, components that can detect failure and automatically switch over to redundant components
Example	e.g.: Non-critical software and IT services (Amazon etc.)	e.g.: Critical applications (related to healthcare, defense etc.)

3.3 Implementation of Fault Tolerance

- Hardware systems backed by equivalent components. Replicate data and compute functions between redundant servers
- Build fault-tolerance into network architecture. Multiple paths between nodes in a data-center, or mechanisms to handle link failure and switch failure
- Handle link failures transparently without affecting cloud functionality. Avoid forwarding packets on broken links.
- Redundant power supply using generators.
- Software instances are made fault tolerant by setting up duplicate instances. (e.g. if DB is running and it fails, switch to another instance of the same DB running on another machine)

3.3.1 Chaos Monkey

- Chaos Monkey is a suite of tools designed by engineers at Netflix designed to randomly introduce failures in a production environment.
- Chaos Monkey is used to purposefully introduce faults into systems that are under development so that fault tolerance can be integrated as early as possible and tested at any time.

- By regularly "killing" random instances of a software service, it is possible to test a redundant architecture to verify that a server failure does not noticeably impact customers.
- Chaos Monkey relies on Spinnaker (an open source CI/CD tool similar to Jenkins) that can be deployed on all major cloud providers (AWS, Google App Engine, Azure)
- The suite of tools developed by Netflix under this includes:
  - **Chaos Kong**: Drop an entire AWS Region
  - **Chaos Gorilla**: Drop an entire AWS Availability Zone
  - **Latency Monkey**: Introduces delays to simulate network outages or congestion
  - **Doctor Monkey**: Monitor performance metrics, detect unhealthy instances, for analysis of root causes and eventual fixing/retirement of the instance.
  - **Janitor Monkey**: Identify and clean unused instances
  - **Conformity Monkey**: Identifies non-conforming instances according to a set of rules (age of instances, security groups of clustered instances etc.)
  - **Security Monkey**: Search for and delete instances with known security vulnerabilities and invalid config
  - **10-18 Monkey** Detects problems with localization and internationalization for software serving customers across different geographic regions.

3.4 Fault Tolerant Design Patterns for Microservice Architectures

3.4.1 Use Asynchronous Communication

- Avoid long chains of synchronous HTTP calls when communicating between services as the incorrect design leads to major outages.
- This helps minimize ripple effects caused by network outages.

3.4.2 Work around Network Timeouts

- To ensure that reosurces are not indefinitely occupied, use network timeouts.
- Clients should be designed not to block indefinitely and to always use timeouts when waiting for a response

3.4.3 Retry with Exponential Backoff

- Perform retries to service calls at exponentially increasing intervals.
- This is done to counter intermittent failures when the service is only unavailable for short time.
- Microservices should be designed with circuit breakers so that the increased network load due to the successive retries does not cause Denial of Service (DoS)

3.4.4 Circuit Breaker Pattern

- In this approach, the client process tracks the number of failed requests.
- If the error rate exceeds a configured limit, a "circuit breaker" trips so that further attempts fail immediately. (large number of failed requests implies that the service is unavailable, hence don't contact now)
- After a timeout period, the client should try again and, if the new requests are successful, close the circuit breaker.

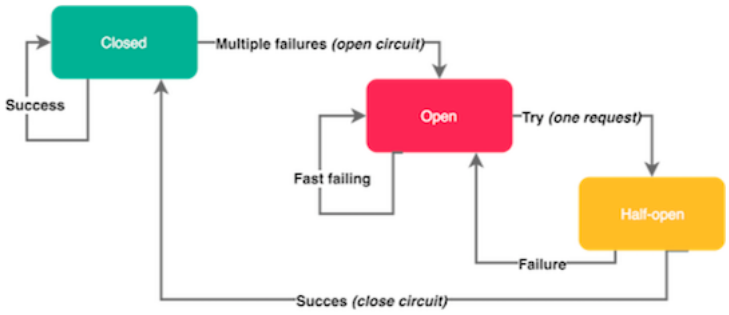

Figure 12: Circuit Breaker Design Pattern

3.4.5 Fallback Mechanisms

- Fallback provides an alternative solution during a service request failure.
- Fallback logic is implemented that runs when a request fails. The logic may involve returning a default value, or returning some cached value.
- Fallback logic must be simple and failure-proof as it is itself running due to a failure.

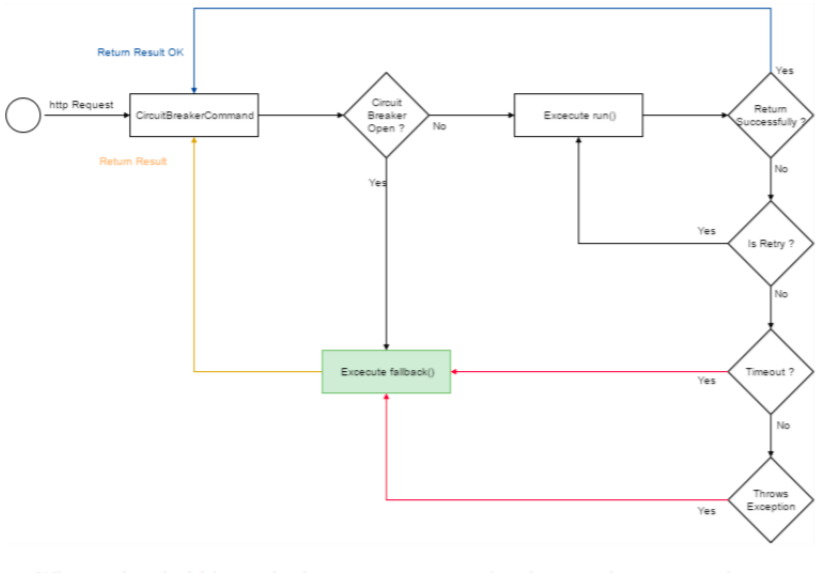

Figure 13: Circuit Breaker Pattern with fallback logic

3.4.6 Limit number of queued requests

- Clients impose an upper bound on number of outstanding requests that can be sent to a service.
- In case this upper bound is crossed, the remaining requests should automatically fail.
- This is a form of rate limiting or throttling, controlling the rate of requests sent in a time period
- If request arrival rate exceeds the processing rate, the incoming requests can either be queued in a FIFO queue, or discarded when the queue fills up.
- When the service has capacity, it retrieves messages from this queue and processes them. When the request rate is greater, the available capacity messages are processed in order and are not lost.

4 Task Scheduling Algorithms

- Policies that assign tasks to the appropriate available resources (CPU, Memory, bandwidth) in a manner that ensures maximum possible utilization of those resources
- Categorized into:
  - **Immediate** scheduling algorithms assign tasks to VMs as soon as they arrive
  - **Batch** schedulers group tasks into batches and schedule individual batches onto VMs
  - **Static** schedulers do not take into account the current state of the system, but rather use only prior available information on the state. Divides all traffic equally among all VMs, e.g. round robin or random scheduling.
  - **Dynamic** schedulers take into account the current system, do not need any prior information on the system, and distribute tasks as per the relative capacities of the VMs
  - **Preemptive** scheduling means that tasks can be interrupted and moved to other resources where they can continue execution
  - **Non-preemptive** scheduling means that a task cannot be reallocated to a new VM until its execution is complete
- Levels of task scheduling:
  - The **Task level**. Consists of tasks or cloudlets sent to the system by the users
  - The **scheduling level**. It is responsible for mapping tasks to resources to get highest resource utilization with minimum completion time for all tasks (aka minimum makespan)
  - The **VM level**. Consists of VMs that execute the scheduled tasks

4.1 FCFS Scheduling

- Advantage: Simplest to implement and understand.
- Drawback: Leads to longer wait times and lower resource utilization
- Algorithm: Assign tasks to VMs in the order of their arrival time. If more tasks than VMs then assign tasks in a round-robin manner.

4.2 SJF Scheduling

- Advantage: Lowest average wait time among all algorithms (provably optimal).
- Drawback: Long tasks are forced to wait for long times (starvation), and cannot be implemented at the short-term level.
- Algorithm: Sort all available tasks in increasing order of their execution time. Then assign the tasks to VMs in sequential order of the VMs.

4.3 Min-Max Scheduling

- Advantage: Efficient resource utilization
- Drawback: Leads to increased waiting time for small and medium tasks
- Algorithm: Sort tasks in decreasing order of execution time (longest task first). Sort VMs in decreasing order of performance (most powerful VM first, i.e. VM with minimum latency first). Now assign tasks to VM in order.

5 Cluster Coordination

- Consensus is the task of getting all processes in a group to agree on a single value based on votes gathered from all processes. The value agreed upon has to be submitted by one of the processes, i.e. this value cannot be invented by the consensus algorithm
- Synchronous processes are those that follow a common clock, while asynchronous processes are those where each process has an individual clock.
- In asynchronous systems, it is not possible to build a consensus algorithm as it is impossible to distinguish between processes that are dead, and those are just slow in responding.
- If even one process crashes in an asynchronous system, then the consensus problem is **proved** to be unsolvable in this paper by Fischer, Lynch and Patterson from 1985.
- Why solve consensus? It is important because many problems in distributed computing take a similar form to the consensus problem such as:
  - Leader election
  - Perfect failure detection
  - Mutual exclusion (agreement on which node gets access to a particular shared resource)
- The **properties** to be satisfied by asynchronous consensus are:
  - **Validity**: The system cannot accept any value that was not proposed by atleast one node. If every node proposes the same value then that value is accepted
  - **Uniform Agreement**: No two correct processes can agree on different values after a single complete run of the algorithm
  - **Non-Triviality/Termination**: All the processes must eventually agree on a single value

5.1 Consensus Algorithm: Paxos

5.1.1 Roles in Paxos

- Every node in a Paxos system is either a **proposer**, a **acceptor** or a **learner**.
- **Proposers** try to convince the acceptors that the value proposed by them is correct
- **Acceptors** receive proposals from proposers. They also inform the proposer in the event that a value other than the one proposed by them was accepted
- **Learners** announce the outcome of the voting process to all the nodes in the distributed system.

5.1.2 Paxos Phase 1 (Prepare - Promise)

- Every proposer creates a **prepare message** containing the node ID of the proposer (node ID is a single positive integer, monotonically increasing and unique to a single node), as well as the value proposed.
- This message is sent to a majority of the acceptors.
- If the acceptor has never seen any prepare message before, it sends back a **prepare response** message which indicates that the acceptor promises not to accept any prepare with an ID less than the current one.
- In case the acceptor has seen a message before, it compares the ID in the incoming prepare message with the max ID it has seen.
  - If current ID is greater than max ID, then it sends back a **prepare response**. This promise message contains the Id and value of the highest previously accepted message. The promise is made not to accept any prepare message with ID less than current ID.
  - If current ID is smaller than max ID then simply ignore the current prepare message

5.1.3 Paxos Phase 2 (Propose - Accept)

- Once a proposer receives a prepare response from a majority of the acceptors, it can start sending out accept requests.
- A proposer sends out an accept request containing its node ID, and the highest value it recevied from all the prepare responses.
- If an acceptor receives an accept request for a higher or equal ID than it has already seen, it accepts and sends a notification to every learner
- A value is chosen by the Paxos algorithm when a learner discovers that a majority of acceptors have accepted a value.

5.2 Leader Election Algorithms

- Elect a single leader from a group of non-faulty processes such that all processes agree on who the leader is
- Leader election criteria:
  - Any process can call for an election, but a process can call for only one election at a time
  - Multiple processes can call an election simultaneously. The result of all these elections is the same
  - Result of an election does not depend on which process called the election
- Conditions for successful leader election run:
  - **Safety**: Every non faulty process p must either elect a process q with the best attribute value, or NULL
  - **Liveness**: An election, once started, must terminate, and the result of a terminated election cannot be NULL.
- Each node has an attribute value which is its identifier. The value determines that node's fitness for leadership. (e.g.: CPU power, disk space, etc.)

5.2.1 Ring Election

- All N nodes are arranged in a logical ring. The node $p[i]$ has a direct link to the next node $p[(i+1) mod N]$
- Messages are only sent in clockwise order around the ring. The node that discovers the failure of the existing coordinator starts the process of election by sending its attribute value in an **election** message to the next node.
- Algorithm:
  - When a node receives an election message, it checks the attr value in that message.
  - If the incoming attribute value is less than the node's attribute value, the node overwrites the message with its own attribute value and sends it to the next node in the ring.
  - If the incoming attribute value is greater than the node's attribute value, the node simply forwards the message to the next node in the ring.
  - If the incoming attribute value happens to the same as the node's attribute value, the election stops. The newly elected leader sends out **elected messages** to the next node, which forwards them to the next node and so on until all nodes in the ring have received an elected message containing the leader's ID.
- Worst case occurs when the leader is the anti-clockwise neighbour of the initiator. In this case the *number of messages exchanged* is $3N - 1$ ($N - 1$ election messages to reach the leader, N election messages to confirm that no higher node exists and finally N elected messages)

- The simple ring election algorithm offers safety and liveness as long as nodes don't crash during election.
- If the nodes crash during election, then it could lead to an election message going around the ring infinitely, thus the election goes on forever and liveness is not followed.

5.2.2 Modified Ring Election

- Similar set up to the simple ring election case.
- Algorithm:
  - The initiator sends out an **election message** to the next running node in the ring. The first message contains the attr value of the initiator.
  - If a node receives an election message, it simply appends its own attribute value to the message and forwards it back.
  - When the election message completes one round and reaches the initiator again, the initiator selects the process with the best attribute value and crafts a **coordinator message**.
  - The **coordinator message** is of the form $coord(n_i)$ where n_i is the elected node value. The coordinator message is sent around the ring, each node appends its own attribute value to the end of the message
  - Once the coordinator message completes one round and reaches the initiator again, the initiator checks if the value in the coord message is there in the appended list of IDs. If it is there then election stops. If not then the initiator once again starts the election.
- Supports concurrent elections – an initiator with a lower id blocks election messages by other initiators
- If a node fails, then the ring can be re congifured to make it continuous again if all nodes in the ring know about each other.
- If the initiator is not faulty, then *message complexity* $= 2N$, *turnaround time* $= 2N$ and message size grows as $O(N)$

5.2.3 Bully Algorithm

- Modified ring leader election algorithm is not suitable for asynchronous systems where there is no upper bound on message delays, meaning there can be arbitrarily slow processes.
- This is because of the fact that a process p_i may not respond to the election messages as it is slow (but not failed), and the slow initiation and reorganization (in case of node failure)
- In the **Bully Algorithm**, every process is aware of the Process ID (PID) of every other process. The algorithm is summarized as follows:
  - A process P initiates an election by sending **election messages** to processes that have a *higher PID* than it. If there is no response to the election message within a timeout, then the election is done, the process that sent the election messages is the leader.
  - If P receives a reply to its election message, then P waits for the corresponding coordination message from the higher PID process. If this does not arrive within a timeout, then the election is restarted.
  - At the end of this election message exchange, there is a process P_l that knows for sure that it has the highest PID. P_l sends a **coordination message** to the processes with *lower PID* than it. This is the end of the election.
- Worst case: message overhead is $O(N^2)$, turnaround time is 5 message times
- Best case: $N - 2$ message overhead, turnaround time is 1 message time.

6 Distributed Locking

- A lock is a mechanism that allows multiple processes/threads to access shared memory in a safe manner avoiding race conditions. Locks are implemented as semaphores/mutexes/spinlocks.
- Locks are operated in the following sequence:
  - Acquire the lock. This gives the process sole control over the shared resource
  - Perform the tasks needed on the shared resource
  - Release the lock. THis gives the other waiting processes a chance to access the shared resource
- A distributed lock is one that can be acquired and released by different nodes (instead of processes and threads on only one node).
- Advantages of distributed locking:
  - **Efficiency**: prevents expensive computations from happening multiple times.
  - **Correctness**: Avoid inconsistency, corruption or loss of data.
- Features of distributed locks
  - **Mutual exclusion**: Only one process can hold a lock at a given time
  - **Deadlock-free**: locks must be held and released in a manner that avoids deadlocks between processes. No one process can hold a lock indefinitely, locks are released after a certain timeout.
  - **Consistency**: Despite any failover situation caused by a node failure, the locks that the original node held must still be maintained.

6.1 Types of Distributed Locks

6.1.1 Optimistic Locks

- Useful for stateless environments where there is low amount of data contention.
- Makes use of version numbers to maintain consistency, instead of locks.
- Use a version field on the database record we have to handle, and when updateing it, check if the data read has the same version of the data being written.

6.1.2 Pessimistic Locks

- Involves the use of an intermediate single **lock manager**(LM) that allows nodes in the system to acquire and release locks. All acquire and release operations go through the DLM
- The LM becomes a single point of failure. If the LM crashes then no node can acquire a lock hence no operations can be performed.
- A node can perform a database transaction only after acquiring the lock from the LM. While the lock is held by a node, the LM refuses all acquires from any other nodes. After the transaction is done, the node releases its held lock.
- Expiration time is set on all locks so that no lock is held indefinitely. If this timer expires before the process finishes the task, then another process acquires the lock and both processes release their locks hence inconsistency.
- The above is resolved using fencing

6.2 Fencing

- Everytime the LM grants a lock (in response to an acquire) it sends back a **fencing token** to the client.
- Along with every write request to the DB, the client sends this fencing token.
- If the DB has processed a write request with token ID N then it will not process write requests containing token ID less than N
- Token ID less than N indicates that the node had acquired the lock earlier but the timeout has expired hence that lock is not valid anymore

6.3 Distributed Lock Manager

- The DLM runs on all the cluster nodes, each having an identical copy of the same database.
- DLM provides software applications running on a distributed system with a means to synchronize their accesses to shared resources.
- The DLM uses a generalized concept of a resource, which is some entity to which shared access must be controlled.

7 Zookeeper

- ZooKeeper is a service for coordinating processes of distributed applications
- Zookeeper offers a hierarchical key-value store, to provide a distributed configuration service, synchronization service, and naming registry for large distributed systems.
- Useful for lock management (keep it outside of programmer's hands for best results) and avoiding message based control (in async systems message delivery is unreliable)
- Zookeeper maintains configuration information, perform distributed synchronization and enable group services.
- **Properties** of Zookeeper:
  - Simple: leads to high robustness and fast performance
  - Wait-free: slow/failed clients do not interfere with needs of properly-functioning clients (interactions are loosely coupled)
  - High availability, high throughput, low latencies
  - Tuned for workloads with high % of reads
  - Familiar interface

7.1 Uses of Zookeeper

- As a **naming service** similar to DNS but for nodes in a distributed system
- **Configuration management** : latest configuration information of the system for a joining node.
- **Data consistency** using atomic operations
- **Leader election**
- **Distributed Locking** to avoid race conditions
- **Message queue** implementation

7.2 Advantages

- Simple distributed coordination
- Synchronization can be implemented
- Ordered messages
- All data transfers are **atomic** (i.e. no partial data transfer, either full or none)
- Reliability through replication of data

7.3 Disadvantages

- Less feature-rich compared to other such services (e.g. Consul which has service discovery included)
- Dependence on TCP connection for client-server communication
- Fairly complex to understand and maintain

7.4 Working of Zookeeper

7.4.1 Znode

- Zookeeper essentially provides a stripped-down version of a highly available distributed file system. The hierarchy of this "file system" is made up of objects called *znodes*.
- A znode acts as a container of data (like a file) but also as a parent to other znodes (like a directory).
- Each znode can store upto 1 MB of data. The limited amount is because Zookeeper is used for storing only config information (status, networking, location etc.)
- Every znode is identified by a name, which is a path separated by /, with the root node having the path as just '/'.
- A znode with children cannot be deleted.
- Znodes can be:
  - **Ephemeral**:
    - Let a client C_1 create a znode. If C_1's connection session with the server end, then the ephemeral znode created by C_1 will also be destroyed.
    - Ephemeral znodes are visible to all clients despite their lifetime being tied to a single client
    - Ephemeral znodes cannot have children.
  - **Persistent**:
    - A persistent znode continues to stay in the database until and unless a client (not necessarily the creator of the znode) explicitly deletes it.
  - **Sequential**:
    - Zookeeper assigns a sequential ID as part of the name of the znode, whenever a sequential znode is created.
    - The value of a monotonically increasing counter (maintained by the parent znode) is appended to the name of the newly created one.
    - Sequentially numbered znodes enforce a global ordering on the events in a distributed system. Simple locks can be built using sequentially numbered znodes.

7.4.2 Working

- Each Zookeeper server maintains an in-memory copy of the data tree that is replicated across all the servers.
- Only transaction logs are kept in a persistent data store for high throughput
- Each clients connects to a single Zookeeper server using a TCP connection. Client can switch to another Zookeeper server if the current TCP connection fails.
- All updates made by Zookeeper are totally ordered. The order is maintained by the use of the *zxid* or Zookeeper Transaction ID.
- Distributed synchronization is maintained using Zookeeper Atomic Broadcast or ZAB Protocol.
- A client can **watch** a znode, meaning that when any changes are made to the watched znode, the client receives a notification.

7.5 Use Cases

7.5.1 Leader Election

- Client creates persistent znode called `/election`. All clients watch for children creation/deletion under this `/election` znode.
- Each server that joins the cluster tries to create a znode called `/election/leader`. Only one server succeeds in doing this, and that server is elected as the leader
- All the servers call `getChildren("/election")` to get the hostname associated with the child node `leader`, which is the hostname of the leader.
- As the `leader` znode is ephemeral, if the leader crashes then that znode is automatically deleted by the Zookeeper server. This delete operation triggers the watch on the `/election` znode as one of its children has been destroyed
- All the servers who were watching the `/election` znode are triggered. These servers once again repeat the same process and a new leader is elected

7.5.2 Distributed Locking

- Similar algorithm to leader election, with `/lock` used as the parent instead of `/election` (name different dasall)
- The client that successfully creates `/lock` acquires the lock, performs the operation and then destroys `/lock`. Thus the lock is released

7.5.3 Queues and Priority Queues

- A znode called `<path>/queue` is designated to hold the queue.
- Insertion into the queue is done by creating an **ephemeral and sequential** znode of the form `<path>/queue/queue-X` where X is a sequence number assigned by Zookeeper.
- Deletion is done by calling `getChildren()` on the `/queue` znode and processing the list obtained from the lowest sequence number first.
- Priority queue implementation is different only in 2 ways:
  - Always use up-to-date list of children. Invalidate all old children lists as soon as watch notification is triggered
  - Queue node names end with `/queue/queue-YY` where `YY` represents the priority (lower value means higher priority)

7.6 Alternatives to Zookeeper

- **Consul**: Service discovery and configuration tool, highly available and scalable
- **etcd**: Distributed key-value store, open source, tolerates failures during leader election
- **Yarn**: parallelize operations for greater throughput and resource utilization
- **Eureka**: a REST based service used for locating load-balancing and failover services for middle-tier servers
- **Ambari**: Provision, manage and monitor Hadoop clusters. Intuitive web UI backed up by RESTful operations.