Unit-4

Cloud Computing – Cloud Controller (Recap)

We have been studying Cloud computing as an ubiquitous, convenient, on-demand network accessible shared pool of applications and configurable resources that can be rapidly provisioned and released with minimum management effort.

These shared pools of applications and resources are supported by physical IT components, platforms, and software housed in distributed data centers, accessible through the Internet.

We studied different Application architectures and service models like IaaS, PaaS, SaaS, by which these applications and resources hosted in a distributed infrastructure are delivered to customers and users over the Internet as a service at different levels of abstraction.

We also studied different deployment models like public, private, or hybrid, where the resources are deployed by service providers as a shared public platform or shared private platform or a combination of both.

Additionally, we discussed technologies like virtualization supporting the abstraction, sharing, and effective utilization of resources, both compute and storage, to provide reliable and available services as mentioned above to customers.

Cloud Computing – Cloud Controller

We also discussed that the physical resources are structured or organized and configured as distributed clustered systems in distributed Datacenters in different architectural models.

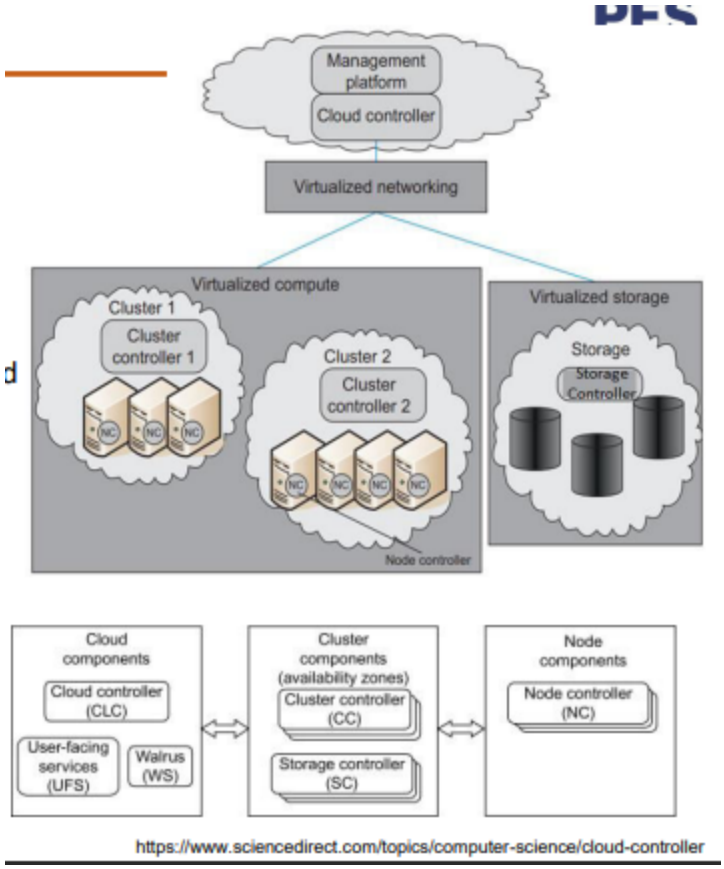
Most of these discussions were focused on technologies, programming models, deployment models, application architectures, and data paths.

Given that customer requests reaching the cloud physical resources will need to be managed, we will need to look at the control path for an application to run in the cloud environment in the next set of classes in Unit 4.

We start with the context of the cloud infrastructure, structured or organized as distributed clustered systems, e.g., as in master-slave, or peer-to-peer distributed system architectures in lecture 37. We discuss challenges associated with these models in terms of unreliable communication in lecture 38, different failures, and their impact on reliability and availability, as well as the fault tolerance needed to support availability in lecture 39.

Various scenarios where distributed transactions will need a consensus mechanism to ensure progress in the presence of certain errors will be discussed in lecture 40 as building consensus. There may be a need for handling scenarios such as if a leader node dies, there is a need for electing a new leader, managing the provisioning and scheduling within the cloud environment, challenges of distributed Locking, and Zookeeper, which will be discussed in subsequent lectures.

Context of Cloud Control - Eucalyptus



Eucalyptus: Eucalyptus is an open-source software suite designed to build private/hybrid cloud computing environments compatible with Amazon Web Service's EC2 and S3.

UFS (User Facing Services): Implements web service interfaces that handle the AWS-compatible APIs. This component accepts requests from command-line clients or GUIs.

Cloud Controller: Provides high-level resource tracking and management. Only one CLC can exist in a Eucalyptus cloud.

Cluster Controller: Manages node controllers and is responsible for deploying and managing VM instances.

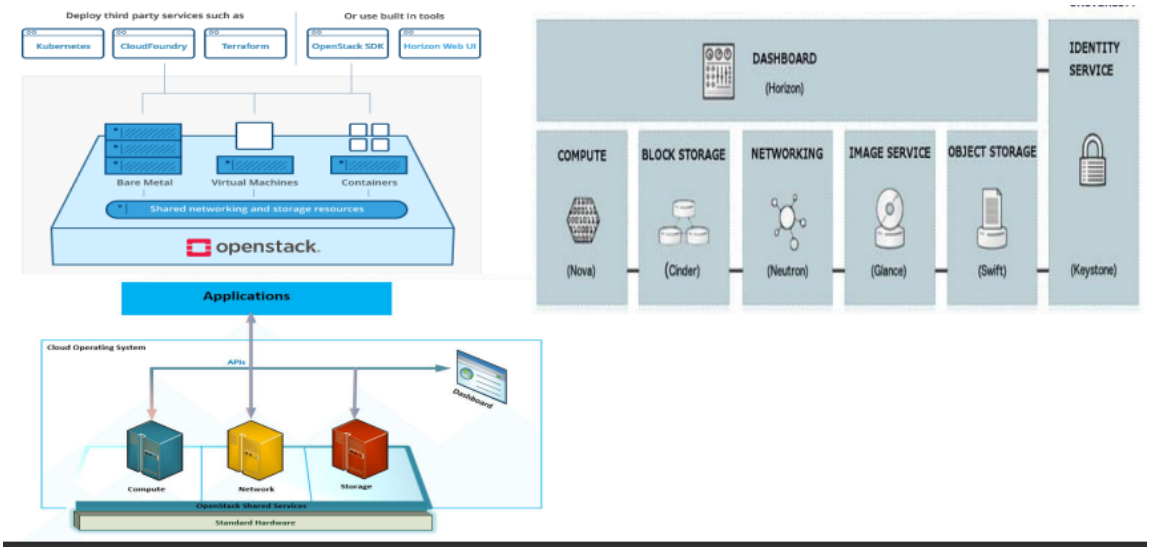
Node Controller: Runs on the machine that hosts VMs and interacts with both the operating system and hypervisors to maintain the lifecycle of instances running on each of the nodes.

Storage Controller: Interfaces with various storage systems.

Cloud Controller Module

- The key interface for users and administrators is the Cloud Controller (CLC), which queries the different nodes and makes broad decisions about the virtualization setup, and executes those decisions through cluster controllers and node controllers.
- The Cloud Controller is a set of web services that provides:
  - System-wide arbitration of resource allocation,
  - Governing of persistent user and system data,
  - Handling authentication and protocol translation for user-visible interfaces.
- A System Resource State (SRS) is maintained with data coming in from CC (Cluster Controller) and NCs (Node Controller) of the system.
- When a user's request arrives, the SRS is used to make an admission control decision for the request based on service-level expectations.
- To create a VM, the resources are first reserved in the SRS and the request is sent downstream to CC and NC. Once the request is satisfied, the resource is committed in the SRS, else rolled back on failure.

Context of Cloud Control - OpenStack



Distributed Systems

A distributed system, also known as distributed computing, is a system with multiple components located on different machines. These components communicate and coordinate actions in order to appear as a single coherent system to the end-user.

Distributed System (Recap)

Depending on the needs of the application, the system components which are part of the cloud infrastructure can be structured, organized, configured, and classified into three different Distributed System Models:

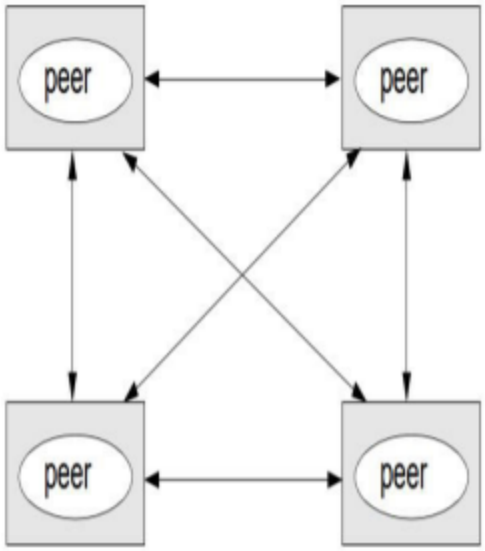
1. **Architectural Models**
    - **System Architecture:**
        - Indicates how the components of a distributed system are placed across multiple machines.
        - Describes how responsibilities are distributed across system components.
        - Example: P2P Model or Client-Server Model
    - **Software Architecture:**
        - Indicates the logical organization of software components and their interactions/independence.
        - Focuses on the components.
        - Example: 3 Tier Architecture
2. **Interaction Models**
    - *How do we handle time? Are there time limits on process execution, message delivery, and clock drifts?*
    - Example: Synchronous distributed systems, Asynchronous distributed systems
3. **Fault Models**
    - *What kind of faults can occur and what are their effects?*
    - Example: Omission faults, Arbitrary faults, Timing faults

Distributed Systems Architecture

Two main architectures:

- **Peer-to-Peer Architecture:**
  - Roles of entities are symmetric.
- **Client/Server or Master-Slave Architecture:**
  - Roles of entities are asymmetric.

Distributed Systems - P2P Systems



In a P2P network, every node (peer) acts as both a client and server.

The system is self-organizing nodes (recognize and create relatively stable connections to other nodes with similar interests/requirements/ capabilities) with distributed control. In other words, no peer machine has a global view of the entire P2P system

- A model of communication where every node in the network acts alike.
- All nodes are equal (no hierarchy)
- A central coordinator is not needed

P2P Architecture

- Peers can interact directly, forming groups and sharing contents (or offering services to each other).
    - At least one peer should share the data, and this peer should be accessible.
    - Popular data will be highly available (it will be shared by many).
    - Unpopular data might eventually disappear and become unavailable (as more users/peers stop sharing them).
- Peers can form a virtual overlay network on top of a physical network topology.
    - Logical paths do not usually match physical paths (i.e., higher latency).
    - Each peer plays a role in routing traffic through the overlay network.

Although P2P model:

- Is Scalable as it is possible to add more peers to the system and scale to larger networks.
- Does not need the individual nodes to be aware of the global state.
- Is Resilient to faults and attacks, since few of their elements are critical for the delivery of service and the abundance of resources can support a high degree of replication.

However, P2P systems are highly decentralized,

- with the individual systems not needing to be aware of the global state,
- thus limiting P2P systems in terms of the ability to be managed effectively and provide security as required by various applications.

This leads to architectures where servers in the cloud are in a single administrative domain and have centralized management, as with Client-Server architecture.

- **Amazon Aurora** – RDBMS built for the cloud with SQL compatibility.
    - Multi-master clusters in Aurora consist of DB instances with both read/write capacity.

    - Multi-master clusters are, however, unconventional since they require different availability characteristics and different procedures for monitoring.

Distributed System - Client-Server Model (Recap)

What is the Client-Server model?
The system is structured where a set of machines called servers (which are performing some process), offer services to another set of machines called clients for their needs.

- The client-server model is usually based on a simple request/reply protocol, implemented with send/receive primitives or using remote procedure calls (RPC).
- The client asks the server for a service, the server does the work and returns a result or an error code if the required work could not be done.
- This organization by its structure distributes the functionality across different machines.
- A Server can provide services for more than one Client and a client can request services from several servers on the network without regard to the location or the physical characteristics of the computer in which the Server process resides.

A master-slave architecture can be characterized as follows:

1. Nodes are unequal (there is a hierarchy).
    - Vulnerable to Single-Point-of-Failure (SPOF).
2. The master acts as a central coordinator.
    - Decision making becomes easy.
3. The underlying system cannot scale out indefinitely.
    - The master can render a performance bottleneck as the number of workers is increased.

Most predominantly undertaken architecture:

- Due to ease of monitoring, to reduce consistency and linearizability issues.

- Shortcoming: Compromise on availability due to single point of failure.

Example: Map Reduce

- Hadoop ecosystem.
- Single Master node.
    - Assigns work to multiple worker nodes.
    - Monitors jobs assigned to workers.
    - Redistributes tasks in case of worker failure.

CLOUD COMPUTING Trouble with Distributed Systems

In a cloud environment where businesses are hosting their services in third-party provisioned distributed cloud infrastructure, it's mandatory for the cloud systems to be reliable and robust. Thus, the cloud controller will need to constantly monitor the infrastructure, including the distributed systems, their communication mechanisms, and their storage, to ensure that these resources are reliable and robust.

These large-scale cloud environments have a lot of commodity hardware, which would mean hardware failures and software bugs can be expected to occur relatively frequently. These hardware failures can trigger other failures, leading to an avalanche of failures that can result in significant outages.

Controlling these cloud resources thus would need an understanding and use of reliability as the metric, and monitoring the cloud components for different categories or types of failures or for unacceptable performances. Monitoring would also need to factor in the challenges related to communication between the nodes in the distributed environment, along with considerations for different categories of failures and approaches to monitor or detect them, and approaches to make the environment more reliable and robust.

This challenge is compounded within a distributed system where:

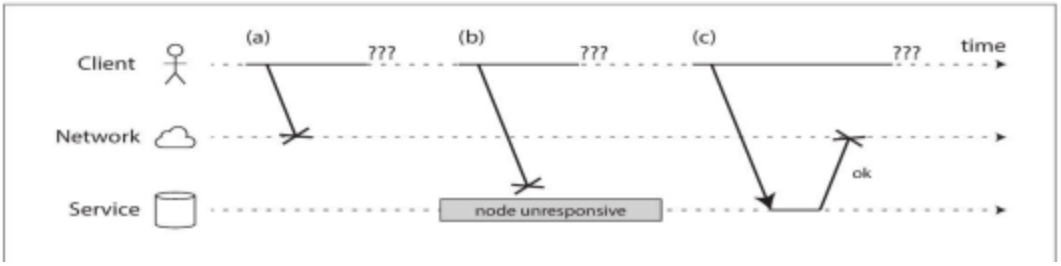
- There may well be some parts of the system that are broken in some unpredictable way, even though other parts of the system are working fine (known as partial failures).

- Given that most distributed systems have multiple nodes in the network, it becomes hard to predict failures, and thus these partial failures become non-deterministic.
- You may not even know whether something succeeded or not, as the time it takes for a message to travel across a network is also non-deterministic.
- This non-determinism and possibility of partial failures is what makes distributed systems hard to work with.

This necessitates mechanisms and components for monitoring different kinds of faults which can occur, building fault models to have ways of handling these partial or full failures, and having mechanisms to tolerate some faults to make these distributed systems reliable and robust.

Consider that you are sending a request and expect a response, where many things could go wrong:

1. Your request may have been lost (perhaps someone unplugged a network cable).
2. Your request may be waiting in a queue and will be delivered later (perhaps the network or the recipient is overloaded).
3. The remote node may have failed (perhaps it crashed or it was powered down).
4. The remote node may have temporarily stopped responding, but it will start responding again later.
5. The remote node may have processed your request, but the response has been lost on the network (perhaps a network switch has been misconfigured).
6. The remote node may have processed your request, but the response has been delayed and will be delivered late.



What do we mean by Reliability?

- Reliability refers to the probability that the system will meet certain performance standards and yield correct output for a specific time.
- It's the ability for a system to remain available over a period of time to support the requirements of the applications under the slated conditions.
- Reliability also means that the application performs the function that the user expected without errors, disruptions, or significant reduction in performance under the expected load and data volume.
- High reliability is extremely important in critical applications. Many applications hosted on the cloud rely on high reliability, as failure or an outage of service can mean the loss of businesses and lives.
- Reliability can be used to understand how well the service will be available in the context of different real-world conditions.

Summarizing Failures and Fault (Recap)

A system or a component is said to "fail" when it cannot meet its promises or meet its requirements. A failure is brought about by the existence of "errors" in the system when the system as a whole stops providing the required service. The cause of an error is called a "fault".

There can be different kinds of faults:

- Transient Faults: Appears once, then disappears.
- Intermittent Faults: Occurs, Vanishes, reappears, but no real pattern (worst form of faults).
- Permanent Faults: Once it occurs, only the replacement/repair of the faulty component will allow the Distributed System to function normally.

A common metric to measure reliability is the Mean Time Between Failures (MTBF), which is the average length of operating time between one failure and the next.

MTBF = Total uptime / # of Breakdowns

For example, if the specified operating time is 24 hours and in that time 12 failures occur, then the MTBF is two hours.

It is impossible to reduce the probability of a fault to zero; therefore, it is usually best to design fault-tolerance mechanisms that prevent faults from causing failures.

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure	A server fails to respond to incoming requests
Receive omission	A server fails to receive incoming messages
Send omission	A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure	The server's response is incorrect
Value failure	The value of the response is wrong
State transition failure	The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Types of Failure from a Designer perspective (Recap)

- Faults can occur both in processes and communication channels. The reason can be both software and hardware.
- Characterization of these different types of faults and fault models is needed to build systems with predictable behavior in case of faults (systems that are fault-tolerant).

Issues which can lead to Faults in a distributed system: Timeouts and Unbounded Delays

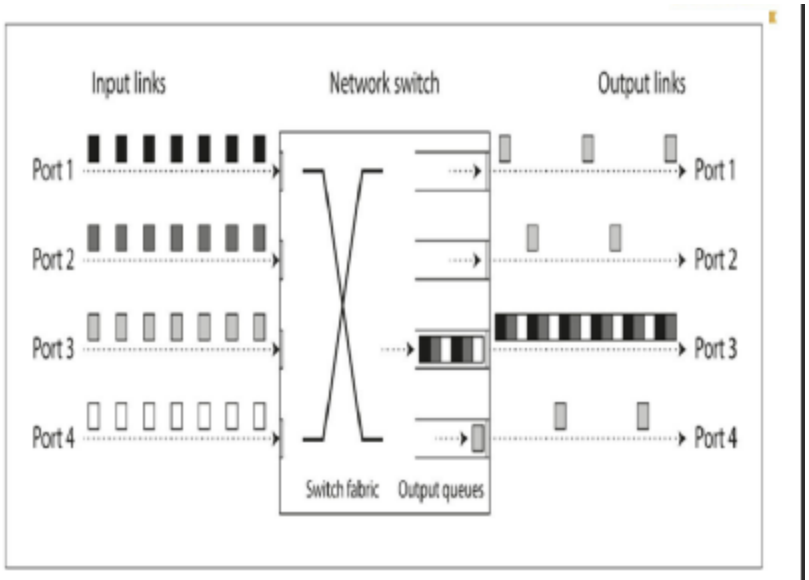
How long should the timeout be?

- Every packet is either delivered within some time d, or it is lost, but delivery never takes longer than d.
- If there is a guarantee that a non-failed node always handles a request within some time r, you could guarantee that every successful request receives a response within time 2d + r.
- If you don’t receive a response within that time, you know that either the network or the remote node is not working.
- Unfortunately, most systems we work with have neither of those guarantees:

- Asynchronous networks have unbounded delays.
- Most server implementations cannot guarantee that they can handle requests within some maximum time.

- Choose timeouts experimentally as there's no "correct" value for timeouts:
  - Measure the distribution of network round-trip times over an extended period, and over many machines, determine the expected variability of delays.
  - Taking into account the application's characteristics, determine an appropriate trade-off between failure detection delay and the risk of premature timeouts.
  - Even better, rather than using configured constant timeouts, systems can continually measure response times and their variability (jitter), and automatically adjust timeouts according to the observed response time distribution.

Issues which can lead to Faults in a distributed system: Network Congestion and Queueing



If several machines send network traffic to the same destination, its switch queue can fill up. Here, ports 1, 2, and 4 are all trying to send packets to port 3.

- On a busy network link, a packet may have to wait a while until it can get a slot (this is called network congestion).
- If there is so much incoming data that the switch queue fills up, there will be a delay in some of the packet being moved to its output links.
- This variability of packet delays on computer networks is most often due to queueing.
- This could also lead to some of the packets being dropped, so these need to be resent—even though the network is functioning fine.

In public clouds and multi-tenant datacenters:

- Increased and effective utilization of resources are achieved using Virtualization and sharing of the resources among many customers (tenants).
- This includes the network links and switches, and even each machine's network interface and CPUs (when running on virtual machines).
- Virtualized multi-tenant resources have overheads which can add latencies towards consuming the network traffic and thus increase Queue lengths.
- This leads to variable delays, congestion, and queueing based on the workload characteristics of the multiple tenants, making it hard for having consistently deterministic and optimized designs.
- Moreover, TCP considers a packet to be lost if it is not acknowledged within some timeout, and lost packets are automatically retransmitted.
- Although the application does not see the packet loss and retransmission, it does see the resulting delay.
- Batch workloads such as MapReduce can easily saturate network links.
- As there is no control over or insight into other customers' usage of the shared resources, network delays can be highly variable if someone near you (a noisy neighbor) is using a lot of resources.

Class of Faults

1. Crash-stop faults:

- The node may suddenly stop responding at any moment, and thereafter, that node is gone forever; it never comes back.
- There are scenarios where this could be detected by other processes/applications, or this may not be detectable by other processes/applications or nodes.

2. Crash recovery faults:

- Nodes may crash at any moment and start responding again after some unknown time.
- In the crash-recovery model, nodes are assumed to have stable storage (i.e., nonvolatile disk storage) that is preserved across crashes, while the in-memory state is assumed to be lost.

3. Byzantine (arbitrary) faults:

- Nodes exhibit arbitrary behavior and may do absolutely anything.

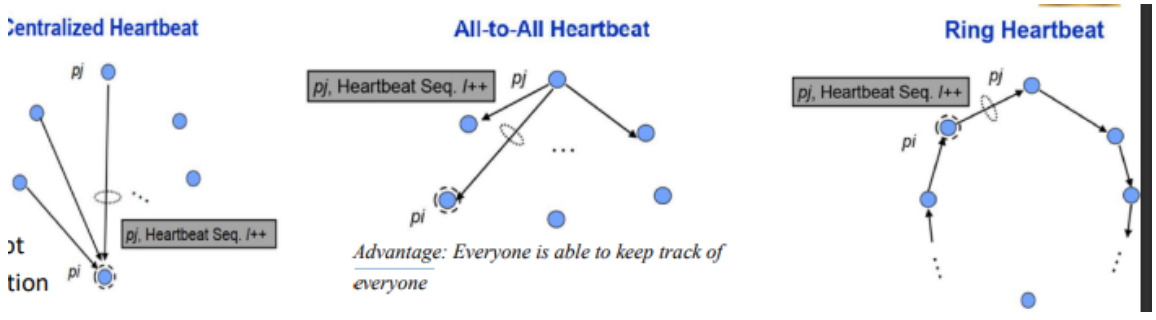
Detection of Failures in a Distributed System

Advantages:

- Everyone is able to keep track of everyone.

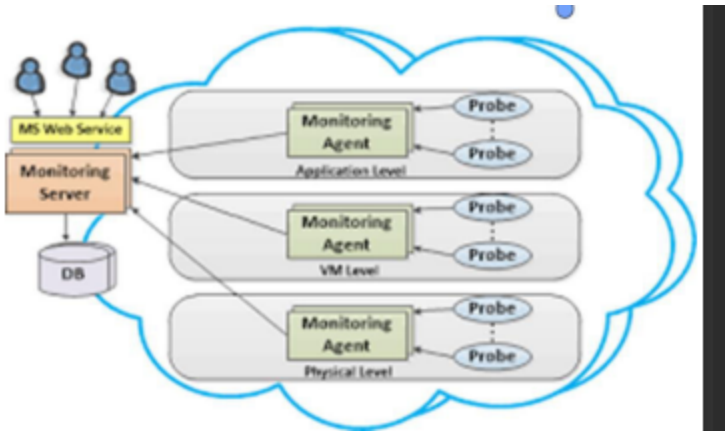
Failure Monitoring: Heartbeats

- Each application periodically sends a signal (called a heartbeat).
- If the heartbeat is not received, the application may be declared as failed.



Probing

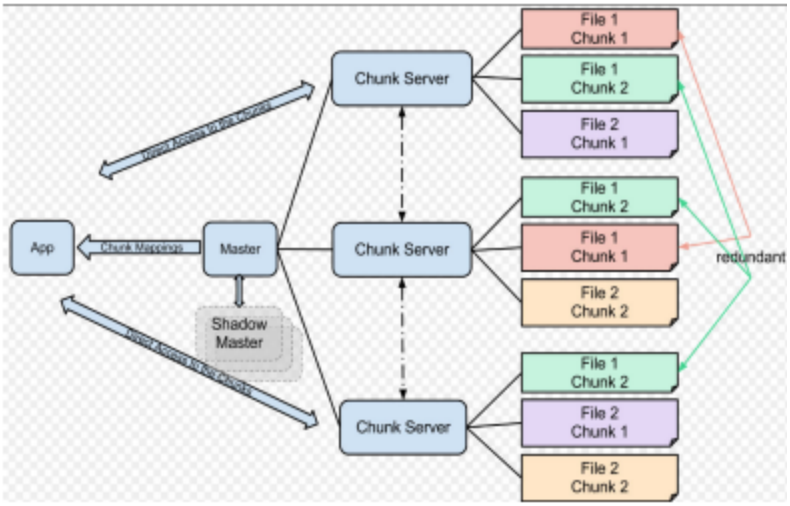
- Periodically sends a probe, which is a lightweight service request, to the application instance and decides based on the response.



Real-World Systems:

1. GFS – Google File System

- Centralized heartbeat mechanism.
- Chunkservers hold the data of the file system.
- Master/Leader node communicates with each chunkserver through Heartbeat messages.
- Will know when a chunkserver is not reachable.
- On chunkserver failure, the Master begins replicating its contents to another chunkserver.
- Heartbeat messages alternate use case: Additional configuration information sent through each heartbeat message at times.



Once a fault is detected: Some strategies for dealing with them

1. **Using asynchronous communication across internal microservices**
    - Long chains of synchronous HTTP calls across the internal microservices can become the main cause of bad outages. Eventual consistency and event-driven architectures will help minimize ripple effects. This approach enforces a higher level of microservice autonomy.
2. **Using retries with exponential backoff**
    - This technique helps to avoid short and intermittent failures by performing call retries a certain number of times, in case the service was not available only for a short time. This might occur due to intermittent network issues or when a microservice/container is moved to a different node in a cluster.
3. **Working around network timeouts**
    - In general, clients should be designed not to block indefinitely and to always use timeouts when waiting for a response. Using timeouts ensures that resources are never tied up indefinitely.

Strategies for dealing with partial failures:

- **Use the Circuit Breaker pattern**

    - 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. After a timeout period, the client should try again, and if the new requests are successful, close the circuit breaker.
- **Provide fallbacks**
    - In this approach, the client process performs fallback logic when a request fails, such as returning cached data or a default value. This is an approach suitable for queries and is more complex for updates or commands.
- **Limit the number of queued requests**
    - Clients should impose an upper bound on the number of outstanding requests that a client microservice can send to a particular service. If the limit has been reached, those attempts should fail immediately. In terms of implementation, the Polly Bulkhead Isolation policy can be used to fulfill this requirement.

Failover Strategy:

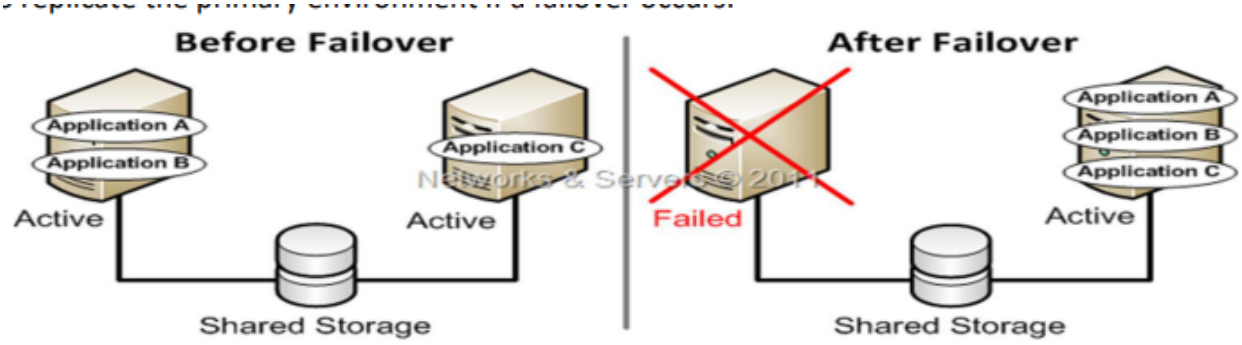
Given that we have replication of data and computes which have been discussed till now, which are available, one of the options once a fault is found would be to failover.

- **Failover** is switching to a replica (this could be a redundant or standby computer server, system, hardware component, or network) upon the failure or abnormal termination of the previously active application, server, system, hardware component, or network.
- Failover can be performed through:
    - An Active-active architecture
    - An Active-Passive or Active-Standby architecture

Active-Active Failover Architecture (Symmetric)

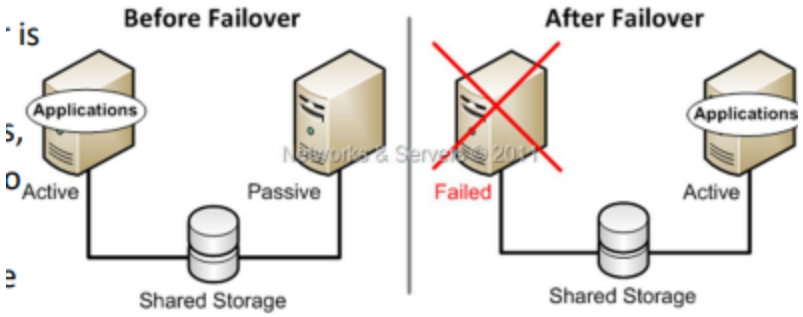
- Each server is configured to run a specific application or service and provide redundancy for its peer.
- In this example, each server runs one application service group, and in the event of a failure, the surviving server hosts both application groups.

- 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.



Active-Passive Failover Architecture (Asymmetric)

- Applications run on a primary, or master, server.
- A dedicated redundant server is present to take over on any failure but apart from that, it is not configured to perform any other functions.
- Thus, at any time, one of the nodes is active, and the other is passive.
- The server runs on the primary node until a failover occurs, then the single primary server is restarted and relocated to the secondary node.
- The failback to the primary node is not necessary since the environment is already running on the secondary node.
- Client connections must then reconnect to the active node and resubmit their transactions.



Note: Failback is the process of restoring operations to a primary machine or facility after they have been shifted to a secondary machine or facility during failover.

VMWare FT

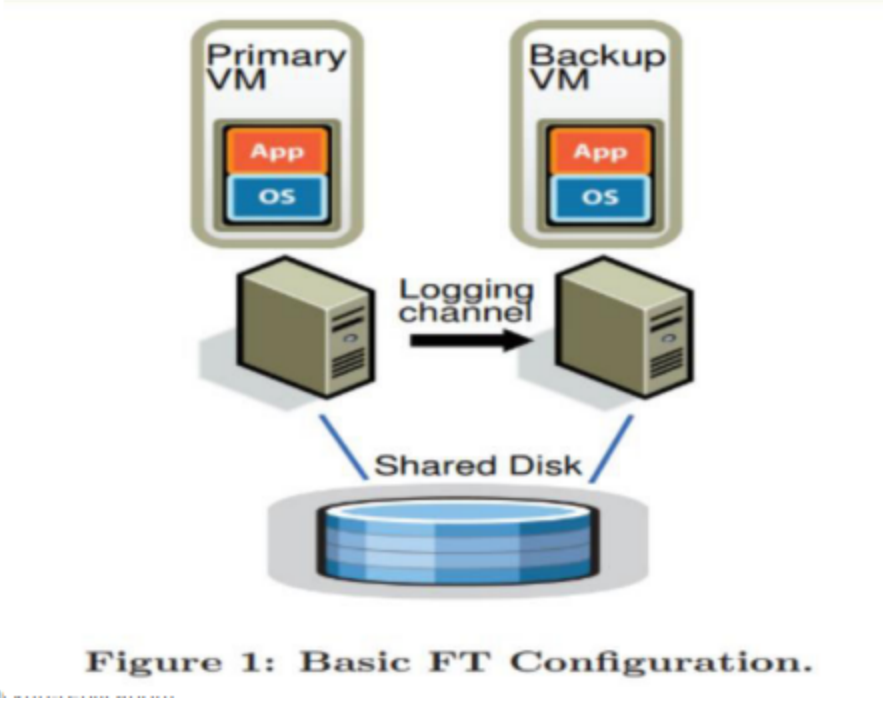


Figure 1: Basic FT Configuration.

VMWare FT (Fault Tolerance) configuration involves Backup and Primary VMs and leverages the concept of virtual lock-step. Here's how it works:

- **Backup and Primary VMs**:
  - VMWare FT operates with two virtual machines: Backup and Primary.

- **Virtual Lock-step**:
  - Both the VMs execute all operations in virtual lock-step.
- **Output to Clients**:
  - Only one VM sends outputs to clients while both are executing the same instructions.
- **Failover**:
  - When the Primary VM fails, the Backup VM takes over seamlessly.
- **Recovery**:
  - After the Primary VM recovers, it acts as the new Backup VM.

VMWare FT ensures continuous operation and minimizes downtime by providing instantaneous failover in case of a Primary VM failure.

Distributed Computing System: Availability

Key aspects of designing a Distributed Computing System beyond Performance are:

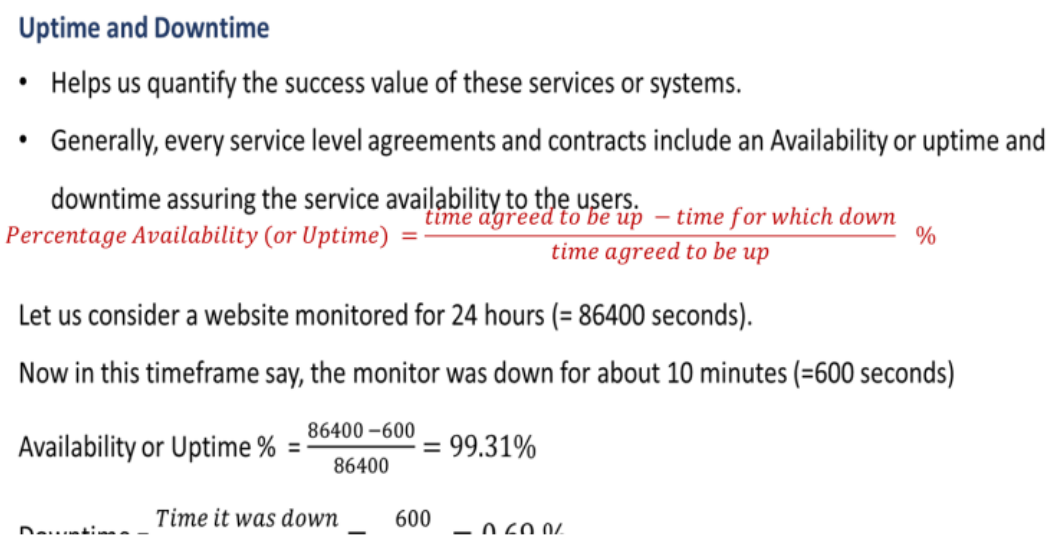
1. **System availability**:
   - Describes the period when a service is available and works as required, when needed during the designated period.
   - Often expressed as a percentage indicating how much time the particular system or component is available in a given period of time, where a value of 100% indicates that the system never fails.
   - It assures users that the system or component they need for their job is available when they need it.
2. **Application flexibility**:
   - Allows applications to use different computer platforms.

System Availability

- **Uptime**:

- Assures that the system will be running or will be functionally operational for the designated time period.
- Typically computed in terms of a percentile, e.g., 99.999%, also called five 9s.

- **Downtime**:
  - Outage duration for which the system would be unavailable or non-operational.
  - The unavailability of the system or network may be due to any system failures (unplanned event), such as a software crash, hardware component crash, or communication failures (network outages).
  - Maintenance routines (planned events) can also lead to unavailability of the system and thereby cause downtime.



Uptime and Downtime

- Helps us quantify the success value of these services or systems.
- Generally, every service level agreements and contracts include an Availability or uptime and downtime assuring the service availability to the users.

$$Percentage\ Availability\ (or\ Uptime) = \frac{time\ agreed\ to\ be\ up - time\ for\ which\ down}{time\ agreed\ to\ be\ up}\ \%$$

Let us consider a website monitored for 24 hours (= 86400 seconds).

Now in this timeframe say, the monitor was down for about 10 minutes (=600 seconds)

$$\text{Availability or Uptime \%} = \frac{86400 - 600}{86400} = 99.31\%$$

$$\text{Downtime} = \frac{Time\ it\ was\ down}{} = \frac{600}{} = 0.69\ \%$$

- **Slow or non-responsiveness**:
  - Refers to a system or service taking unacceptably long periods of time for processing and returning the response to the user (depending on the transaction).

Example

Let's say you monitored a website during 24 hours (which translates to 86,400 seconds), and in that timeframe the website went down for 10 minutes (600 seconds). To define the uptime and downtime percentages, the following calculation is performed:

Total time your website was down: 600 seconds
Total time your website was monitored: 86,400 seconds
Downtime percentage = 600 seconds / 86,400 seconds = 0.0069 = 0.69%
Uptime percentage = 100% - 0.69% = 99.31%

Approaches used by some cloud service providers to measure availability:

- **Microsoft cloud services**:
  - Time when the overall service is up, divided by total time.
- **Gmail**:
  - Percentage of successful interactive user requests for uptime, error rate, and consecutive minutes of downtime for downtime.
- **Amazon Web Services**:
  - Error rate - average percentage of requests that result in errors in a 5-minute interval.
- **Recent published work**:
  - Uses approaches such as "Windowed user-uptime" to factor in user-perceived availability over many windows to distinguish between many short periods of unavailability and fewer longer periods of unavailability.

Fault Tolerance

- **Definition:** Fault tolerance is the system's ability to continue operating uninterrupted despite the failure of one or more of its components.
- **Achievement:** Fault tolerance can be achieved through hardware, software, or a combined solution, such as leveraging load balancers or other components.
- **Purpose:** Fault tolerance enables systems to be reliable and highly available.
- **Approaches to Fault Tolerance:**

- **Fail-Safe Tolerance:** Safety state is preserved, but performance may drop. For example, no process can enter its critical section for an indefinite period due to a failure, or a traffic crossing failure changes the traffic in both directions to red.
- **Graceful Degradation:** Application continues, but in a degraded mode. The extent of degradation is determined by what is considered acceptable. For example, processes will enter their critical sections but not in timestamp order.

Considerations for Building Fault Tolerance

1. **Downtime:** A fault-tolerant system is expected to work continuously with no acceptable service interruption.
2. **Scope or Extent of Fault Tolerance:**
   - Needs to manage failures and minimize downtime.
   - Considers backup components like power supply backups, as well as hardware or software that can detect failures and instantly switch to redundant components.
3. **Cost:**
   - Requires additional cost as it involves continuous operation and maintenance of redundant components.
   - Building fault tolerance requires consideration of the system's need for tolerance to service interruptions, the cost of such interruptions, existing SLA agreements with service providers and customers, as well as the cost and complexity of implementing full fault tolerance.

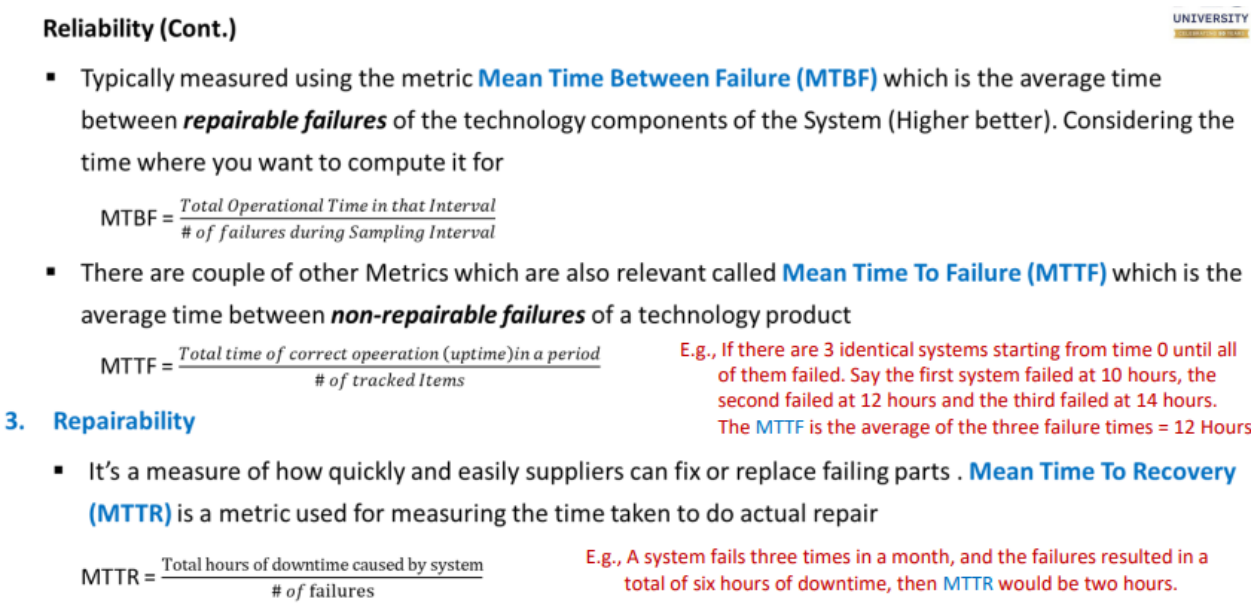
Approaches for Building Fault Tolerance

- The goal is to increase system availability by extending uptime and reducing downtime.
- Fault tolerance supports extending uptime and minimizing possible downtime through different approaches:

1. **Redundancy:**

- Eliminates single points of failure in systems, whether in hardware components (e.g., power supplies, multiple processors, segmented memory, redundant disks) or software components.

2. **Reliability:**
   - Focuses on the dependability of components to function under stated conditions.
   - Analyzed based on failure logs, frequency of failures, etc.
   - MTBF, MTTF, MTTR



Recoverability

- **Definition:**
  - Refers to the ability to overcome a momentary failure in such a way that there is no impact on end-user availability.
  - Includes retries of attempted reads and writes out to disk or tape, as well as the retrying of transmissions down network lines.
  - Involves mechanisms like acknowledgments used to ensure missing or non-ordered delivery of the sequence of data in protocols like TCP.

- Incorporates error control coding (ECC) such as Parity Bit, Hamming Code, CRC, which provide much less redundancy than replication.
- Utilizes techniques like checkpointing and rollbacks, usually through logging, but may not complete tasks predictably in case of rollbacks.

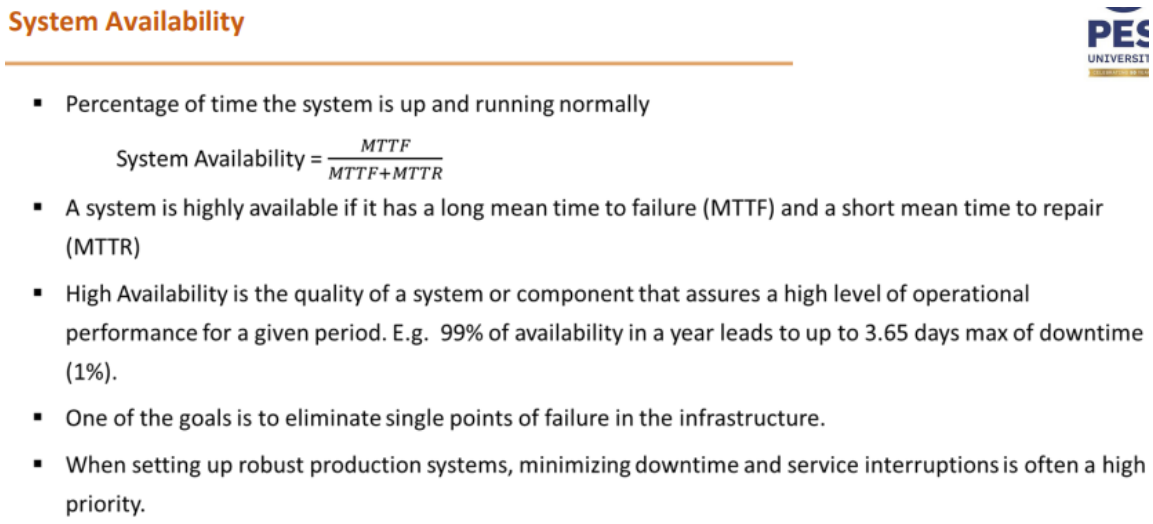
Other Techniques for Fault Tolerance in Distributed Systems

1. **Retries:**
   - Trying the same request again causes the request to succeed, often due to partial or transient failures.
   - Coupled with some kind of exponential backoff.
2. **Timeouts:**
   - Using connection and request timeouts when a request is taking longer than usual helps prevent increased latency in the system and eventual failure.
   - Helps avoid impact on resources used for requests and prevents the server from quickly running out of resources (memory, threads, connections, etc.).
3. **Circuit Breakers:**
   - Stop calling for some time when there is an issue with the frequency of failures, similar to a circuit breaker opening.
4. **Isolate Failures (Bulkheads):**
   - Low coupling and separate processes/threads pools dedicated to different functions ensure that if one fails, the others will continue to function.
5. **Cache:**
   - Save data from remote services to a local or remote cache and reuse the cached data as a response during a service failure.
6. **Queue:**
   - Set up a queue for requests to a remote service to be persisted until the dependency is available.

A few additional techniques used for Fault tolerance in distributed systems

In addition to the generic techniques described earlier, there are different best practices or patterns for fault tolerance in microservices as well.

1. **Asynchronous Communication:**
    - Usage of asynchronous communication, such as message-based communication, across internal microservices.
    - Avoid creating long chains of synchronous HTTP calls across the internal microservices.
    - Eventual consistency and event-driven architectures help minimize ripple effects.
    - These approaches enforce a higher level of microservice autonomy and prevent problems associated with synchronous communication.



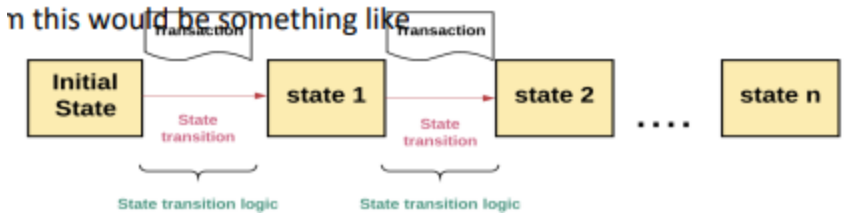
Cluster Co-Ordination Consensus

Distributed Systems and Consensus

- **Replicated State Machine:**
    - Distributed systems are replicated across multiple computers but function as a single state machine using distributed transactions.
    - Each distributed transaction is atomic, meaning the system will either complete entirely or not at all.

- The logic for transitioning from one valid state to the next is called the state transition logic.

- **Consensus in Distributed Systems:**
  - Consensus is achieved when all computers collectively agree on the output value.
  - The replicated state machine ensures that all computers have the same goal, maintained through a consistent transaction log across every computer in the system.
  - The consensus algorithm aims to achieve that all computers are in the same state with a specific value (e.g., f(x)) despite challenges such as faulty computers, broken or slow networks, and the absence of a global clock to synchronize the order of events.



- **The Consensus Problem, Defined:**
  - Consider N nodes, each with:
    - Input variable xp initially set to 0 or 1.
    - Output variable yp initially set to b, which can only be changed once.
  - An algorithm achieves consensus if it satisfies the following conditions:
    - **Agreement:** All non-faulty nodes decide on the same output value (either 1 or 0).
    - **Termination:** All non-faulty nodes eventually decide on some output value or do not decide differently.
  - Additional basic properties or constraints:
    - **Validity:** Any decided value is the proposed value.
    - **Integrity:** A node decides at most once.

        - **Non-Triviality:** There is at least one initial system state that leads to each of the all-0's or all-1's outcomes.

- **A Typical Approach to Consensus:**

Consensus Algorithm Steps:

Step 1: Elect

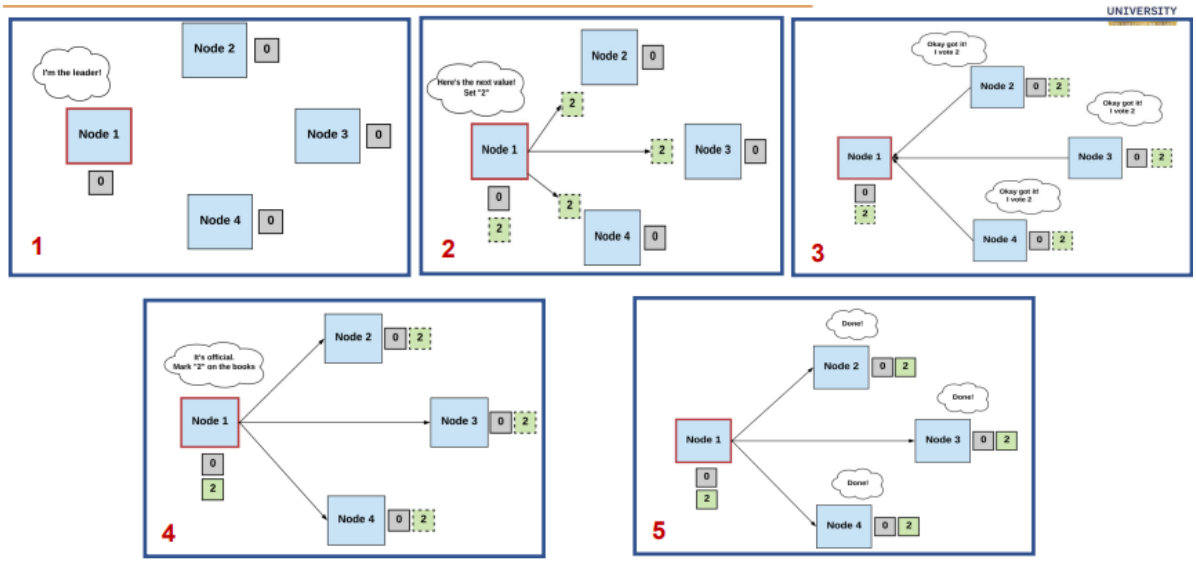
- Processes elect a single process (i.e., a leader) to make decisions.
- The leader proposes the next valid output value.

Step 2: Vote

- Non-faulty processes listen to the value proposed by the leader, validate it, and propose it as the next valid value.

Step 3: Decide

- Non-faulty processes must come to a consensus on a single correct output value.
- If they receive a threshold number of identical votes that satisfy some criteria, the processes decide on that value.
- If consensus is not reached, the steps start over.
- **Summarizing Consensus:**
    - Achieving overall system reliability in the presence of several faulty processes is a fundamental problem in distributed systems, cloud computing, and multi-agent systems.
    - Consensus is the task of all processes in a group to agree on some specific value based on voting, ensuring system reliability during computation.



Common Challenges to be factored in while arriving at consensus

1. **Reliable Multicast:**
    - When a group of servers attempts to solve a particular problem despite failures, they need to reach a consensus.
    - The group of servers together needs to receive the same updates in the same order as sent, known as reliable multicast.
    - Given failures, this leads to the Reliable Multicast problem, which needs to be addressed by consensus.
2. **Membership Failure Detection:**
    - A group membership failure occurs when one of the group members fails or crashes.
    - The problem is for all the remaining group members to be aware of the failure or crashing of one of the group members and be aware of the current status (i.e., whether the process is alive or failed).
    - This is typically called Membership Failure Detection, which is equivalent to consensus.
3. **Leader Election:**

- A group of servers attempts to elect a leader among them, and this decision of electing the leader is known to everyone in that group.
- This problem is termed as leader election.
- Leader election under this specific failure requires the solution of consensus. So, the consensus problem can be solved by anyone, thus leading to a solution for leader election.

4. **Mutual Exclusion:**
    - A group of servers together attempts to ensure mutually exclusive access to critical resources such as writing files in a distributed database.
    - In case of failure, this problem also requires consensus protocols for reliability.

In the consensus problem, every process contributes towards a value, and the goal is to have all the processes decide on some value. Once the decision is finalized, it cannot be changed. There are some other constraints that will also be required to be enforced to solve a particular consensus problem, such as the following:

1. **Validity:**
    - Every process will propose the same value; then, that is what is decided.
2. **Integrity:**
    - Some process must have offered the selected values.
3. **Non-Triviality:**
    - There is at least one initial state that leads to each of all 0s or all 1s outcomes.

Common issues in the consensus problem:

Many problems in distributed systems are either equivalent to or harder than the consensus problem. If the consensus problem can be solved, then those problems in the distributed systems can also be solved. Some of the problems in distributed systems that are either equivalent to or harder than the consensus problem are as follows:

1. **Failure Detection:**
    - If a failure can be detected, the consensus problem can also be solved, or if the consensus problem is solvable, then the faults are also discoverable.
2. **Leader Election:**
    - Exactly one leader is elected, and every alive process is aware of the same. If the leader election problem is equivalent to the consensus problem, then solving the consensus problem is equivalent to solving the leader election problem.
3. **Agreement on a Decision:**
    - In the distributed system, agreement on a decision is hard. Thus, consensus is a fundamental problem, and solving it will be beneficial.

Importance of the consensus problem:

1. **Synchronous Distributed System:**
    - The synchronous model states that each message is received within a limited time, and the drift of each process' local clock has a known bound. Each step in a process belongs between an upper bound and a lower bound. For example, a collection of processors connected by a communication bus such as a Cray supercomputer and a multicore machine.
    - Reaching consensus in a synchronous environment is possible because we can make assumptions about the maximum time it takes for messages to get delivered. Thus, in this type of system, we can allow the different nodes in the system to take turns proposing new transactions, poll for a majority vote, and skip any node if it doesn’t offer a proposal within the maximum time limit. But assuming we are operating in synchronous environments is not practical outside of controlled environments where message latency is predictable, such as data centers which have synchronized atomic clocks.
2. **Asynchronous Distributed System:**
    - An asynchronous model does not have any specified bound on process execution. When the clocks drift, that bound is also arbitrary, and the

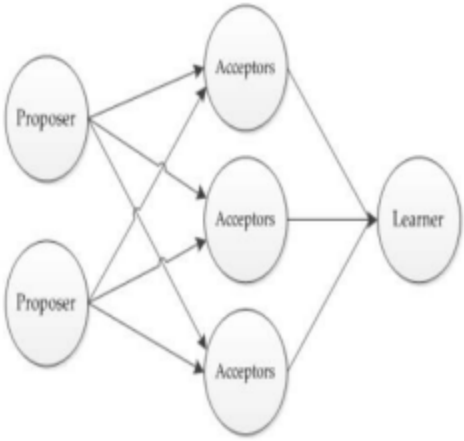
message transmission is delayed; that is, the delay in message receiving is also not bounded. The Internet is a perfect example of an asynchronous distributed system model. Other examples include ad hoc networks and sensor networks.

- In the asynchronous system model, the consensus is impossible to solve. Whatever the protocols or algorithms, there is always a worst possible execution scenario with the failures and message delays, which will prevent the system from reaching the consensus. Therefore, although probabilistic solutions are used to solve the consensus problem, it might be impossible to solve; Fischer–Lynch–Paterson Impossibility says the same.

Paxos Consensus Algorithm

- Family of distributed algorithms used to reach consensus, used by companies like Google and Amazon.
- Defines three roles in a distributed system:
  1. Proposers, often called leaders or coordinators.
  2. Acceptors, processes that listen to requests from proposers and respond with values.
  3. Learners, other processes in the system that learn the final values decided upon.
- Paxos nodes can take multiple roles.
- Paxos nodes must know the number of majority acceptors.
- A Paxos run aims at reaching a single consensus.

Paxos Algorithm



1. The proposer sends a message `prepare(n, v)` to all acceptors.
2. Each acceptor compares `n` to the highest-numbered proposal for which it has responded to a `prepare` message. If `n` is greater than `nv`, then it responds with `ack(nv, v')`, where `v'` is the value of the highest-numbered proposal it has accepted, thus indicating its last received proposal number and value.
3. The proposer waits (possibly forever) to receive `ack` from a majority of acceptors. It then sends `accept(n, max of (v and v'))` to all acceptors (or just a majority).
4. Upon receiving `accept(n, v)`, an acceptor accepts `v` unless it has already received `prepare(n')` for some `n' > n`. If a majority of acceptors accept the value of a given proposal, that value becomes the decision value of the protocol.

- `n`: proposal number
- `v`: proposed value
- `nv`: number of the highest-numbered proposal accepted earlier
- `v'`: value of the highest-numbered proposal it has accepted

Example

- Proposers send two types of messages to acceptors: `prepare` and `accept` requests.
- In the Prepare stage of this algorithm, a proposer sends a `prepare` request to each acceptor containing a proposed value `v` and a proposal number `n`.

- Each proposer's proposal number must be a positive, monotonically increasing, unique, natural number, with respect to other proposers' proposal numbers.

Example

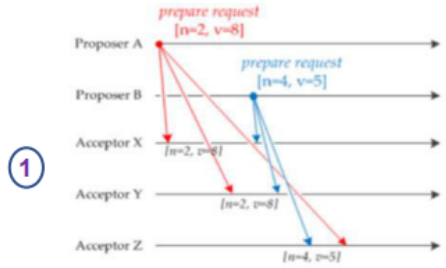
1. Proposer A and B each send prepare requests to every acceptor. In this example, proposer A's request reaches acceptors X and Y first, and proposer B's request reaches acceptor Z first.
    - Each acceptor responds to the first prepare request message that it receives.
    - If the acceptor receives a prepare request and has not seen another proposal, it responds with a prepare response which promises never to accept another proposal with a lower proposal number.

Example

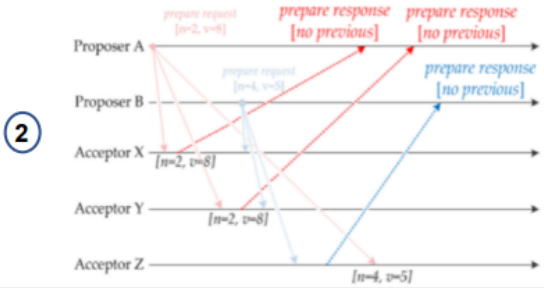
1. Eventually, acceptor Z receives proposer A's request, and acceptors X and Y receive proposer B's request.
    - Acceptor Z ignores proposer A's request because it has already seen a higher numbered proposal (4 > 2).
    - Acceptors X and Y respond to B's request with the previous highest request that they acknowledged and a promise to ignore any lower numbered proposals.
    - Once a proposer has received prepare responses from a majority of acceptors, it can issue an accept request.
    - Since proposer A only received responses indicating that there were no previous proposals, it sends an accept request to every acceptor with the same proposal number and value as its initial proposal (`n=2`, `v=8`), which is rejected as the request number is below 4.
    - Proposer B sends an accept request to each acceptor, with its previous proposal number (4), and the value of the highest numbered proposal it has seen, which is not the earlier 5 but 8 from (`n=2`, `v=8`). So it returns accept values of (`n=4`, `v=8`).
    - A notification is sent to every learner on the accepted value.

Example

- Proposers send two types of messages to acceptors:
  - ***prepare*** and ***accept*** requests.
- In the Prepare stage of this algorithm a proposer sends a ***prepare request*** to each acceptor containing a proposed value v, & a proposal number n.
- Each proposer's proposal number must be a positive, monotonically increasing, unique, natural number, with respect to other proposers' proposal numbers



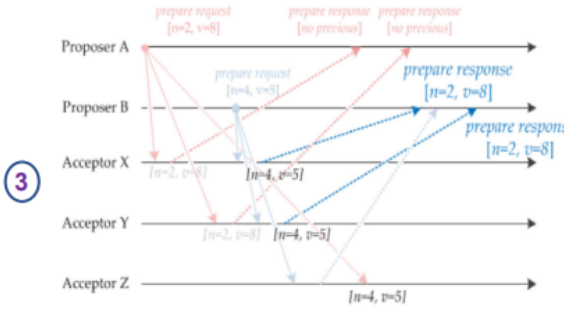
- Proposer A and B each send prepare requests to every acceptor. In this example proposer A's request reaches acceptors X and Y first, and proposer B's request reaches acceptor Z first



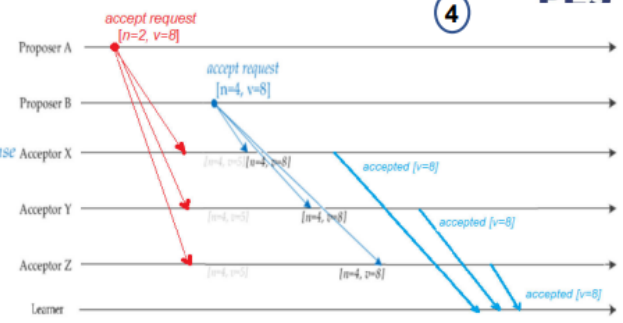
- Each acceptor responds to the first prepare request message that it receives
- If the acceptor receives a prepare request and has not seen another proposal, it responds with a prepare response which promises never to accept another proposal with a lower proposal number.

CLOUD COMPUTING

Example



- Eventually, acceptor Z receives proposer A's request, and acceptors X and Y receive proposer B's request.
- Acceptor Z ignores proposer A's request because it has already seen a higher numbered proposal (4 > 2)
- Acceptors X and Y respond to B's request with the previous highest request that they acknowledged and a promise to ignore and lower numbered proposals



- Once a proposer has received prepare responses from a majority of acceptors it can issue an accept request
- Since proposer A only received responses indicating that there were no previous proposals, it sends an accept request to every acceptor with the same proposal number and value as its initial proposal (n=2, v=8) which is rejected as the request number is below 4
- Proposer B sends an accept request to each acceptor, with its previous proposal number (4), and the value of the highest numbered proposal it has seen, which is not the earlier 5 but 8 from (n=2,v=8). So it returns accept values of (n=4, v=8)
- A notification is send to every learner on the accepted value 8

Cluster Coordination and Leader Election

In many distributed systems, there's a need for a coordinator or leader to manage tasks across different nodes. However, having a single leader can be risky because if the leader fails, the entire system may go down. To solve this problem, we need a mechanism for other nodes to take over the leader role when the current leader fails. This mechanism is called Leader Election.

Why Leader Election?

- Simplifies process synchronization.
- Prevents a single point of failure.

When to elect a leader?

- During system initiation.
- When the existing leader fails.
    - No response from the current leader within a predetermined time.

Goal of Leader Election Algorithm:

1. Elect one leader among non-faulty processes.
2. Ensure all non-faulty processes agree on who the leader is.

Leader Election System Model:

- Any process can call for an election.
- Only one election can be called at a time.
- Multiple processes can call for an election simultaneously.
- The result of an election should not depend on which process calls for it.
- Messages are eventually delivered.

Liveness and Safety Conditions in an Election:

- Liveness: Every node will eventually enter a state that is either elected or not elected.
- Safety: Only one node can enter the elected state and eventually become the leader.

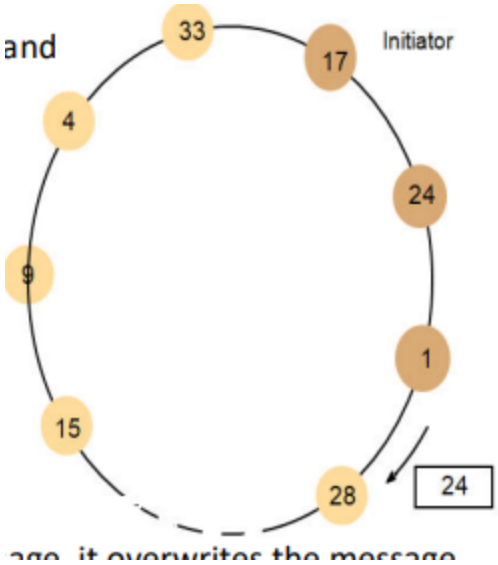
Formal Election Problem:

- Safety: At the end of the election, one non-faulty process with the best attribute value will be elected as the leader, or the election will terminate with no leader (Null).
- Liveness: Every non-faulty process will eventually enter a state that is elected or not elected (Null).

Attributes for Leader Selection:

- Fastest CPU.
- Most disk space.
- Most number of files.
- Priority, etc.

Ring Election Algorithm

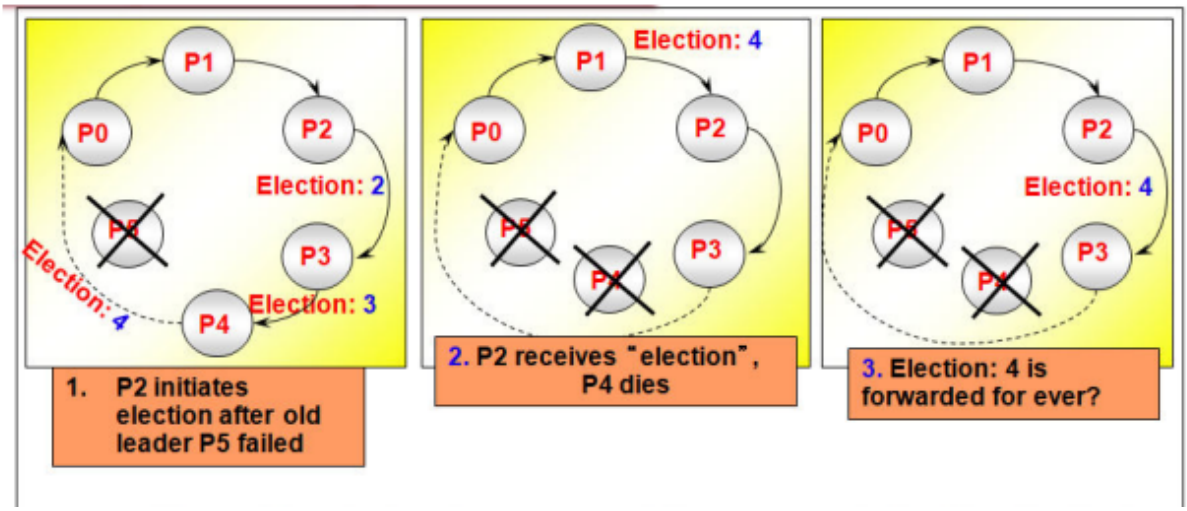


1. **Setup:**
   - N Processes/Nodes arranged in a logical ring.
   - Processes communicate only with their logical neighbors.
   - Messages are sent clockwise around the ring.
2. **Initiating Election:**
   - If a process detects that the old coordinator has failed, it initiates an "election" message containing its own id and attribute.
3. **Handling Election Messages:**
   - When a process receives an election message:
     i. If the received attribute is greater, it forwards the message.

ii. If the received attribute is smaller and the process hasn't forwarded an election message yet, it replaces the message with its own id:attr and forwards it.
iii. If the received id:attr matches its own, it becomes the new coordinator and sends an "elected" message to its neighbor.

4. **Handling Elected Messages:**
   - Upon receiving an elected message:
     i. It updates its variable to store the id of the new coordinator.
     ii. It forwards the message unless it is the new coordinator itself.

Example:



- Process 17 initiates the election.
- The highest process identifier encountered is 24 (final leader will be 33).
- In the worst-case scenario, where the counter-clockwise neighbor has the highest attribute:
  - N-1 messages are required for the new coordinator to be elected.
  - Another N messages are required for the new coordinator to confirm its role.
  - Another N messages are required to circulate the elected messages.
  - Total Message Complexity = 3(N-1).

- Turnaround time = 3(N-1).

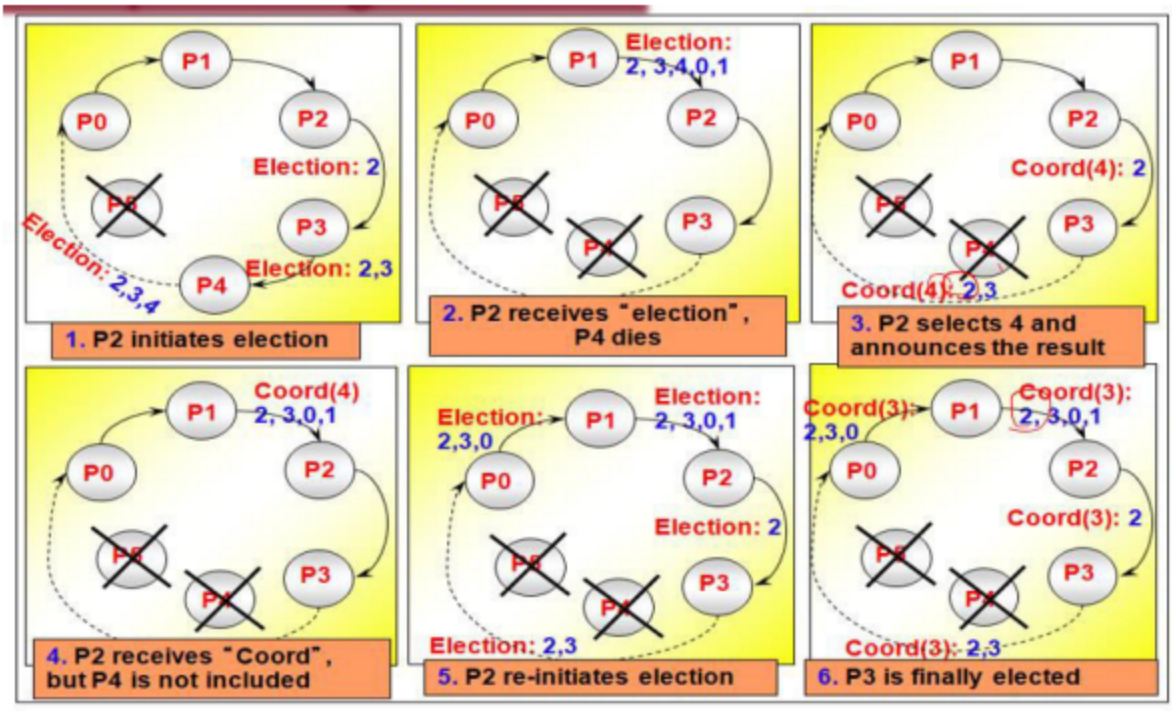
Correctness:

- In the absence of failures, safety and liveness conditions are satisfied.

Failures during Election:

- If failures occur during the election:
  - The algorithm may not be able to guarantee safety and liveness.
  - Failure handling mechanisms may need to be incorporated to ensure correctness.

Ring Election Algorithm



1. Setup:

- Imagine we have a group of computers arranged in a circle, like cars on a roundabout.
- Each computer is connected only to the computers next to it.

- If a computer (let's call it Process) notices that the main coordinator computer has stopped working, it starts a special message called "election."

2. Initiating Election:

- If a computer detects that the main coordinator is down, it starts an "election" message.
- This message travels around the circle, passing from one computer to the next, until it comes back to the computer that started it.

3. Handling Election Messages:

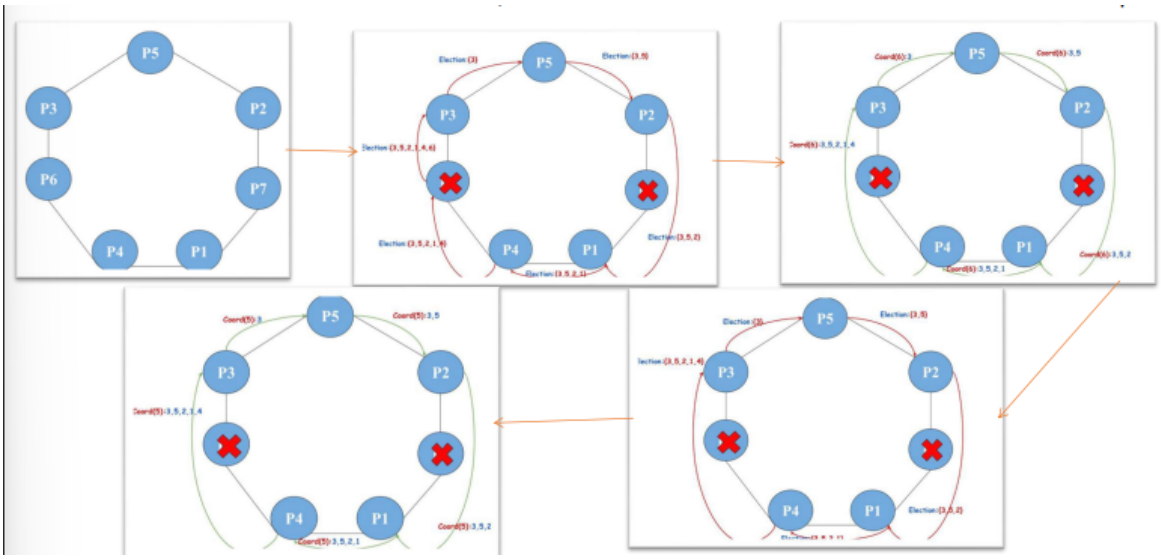
- As the "election" message moves around the circle:
  - Each computer adds its own number and some information about itself to the message.
  - When the message comes back to the computer that started it:
    - The computer looks at the information in the message and chooses the computer with the best number as the new coordinator.
    - It then sends a "coordinator" message with the number of the new coordinator.

4. Handling Coordinator Messages:

- The "coordinator" message also travels around the circle.
- When it comes back to the computer that started the election:
  - If the new coordinator's number is in the list of computers that passed the message, the election is over, and the new coordinator is confirmed.
  - If not, the election process starts again to choose a new coordinator.

Now, let's see how this works in practice:

Problem:



- Process 7 fails, and Process 3 starts the election.
- After Process 3 announces Process 6 as the new leader, Process 6 also fails.

Flow of Events:

1. Process 3 starts an election message.
2. The message travels around the circle, with each process adding its number to the message.
3. Process 3 receives the election message again.
4. Process 3 looks at the numbers in the message and selects Process 6 as the new coordinator.
5. Process 3 sends a "coordinator" message with Process 6's number.
6. The "coordinator" message travels around the circle.
7. When it comes back to Process 3, it checks if Process 6's number is in the list.
8. If Process 6's number is in the list, the election is over, and Process 6 is confirmed as the new coordinator.
9. If not, the election process starts again to choose a new coordinator.

Leader election: Bully Algorithm

Leader Election is Hard

Leader election is like a school election where each student wants to be the class monitor. But in a big school, where everyone is in different classes and they can't see each other, organizing the election is a bit tricky.

Why is it called the Bully Algorithm?

Imagine there's a group of friends playing, and one friend, let's call him Bob, is leading the game. If Bob suddenly leaves, then the friend with the highest rank among the remaining ones becomes the new leader. This friend, let's call him the bully, then tells everyone else that they are the new leader. This is why it's called the Bully Algorithm.

- When a process with the next highest ID (after the current leader) detects the leader failure, it elects itself as the new leader.
- It sends a coordinator message to other processes with lower identifiers
- At this point, the election is completed. Therefore, this is called a bully algorithm.

How does the Bully Algorithm work?

1. **Bully's Confidence:** If a process (a computer) knows it has the highest rank, it declares itself the leader and tells everyone.
2. **Initiating an Election:** If a process isn't sure it's the highest ranked, it starts an election by telling only the processes with higher ranks. It waits for a response.
3. **No Response:** If no one responds, it declares itself the leader and tells everyone.
4. **Response Received:** If it receives a response, it knows there's a bully (a process with a higher rank). It waits for that bully to tell everyone it's the leader. If that doesn't happen, it starts a new election.

Timeout Values

- If a process doesn't get a response within a certain time, it assumes the other process is not working.
- It's important to set the timeout value correctly. If it's too short, we might end up with too many elections. If it's too long, we might take too long to find out

there's a problem.

Assumptions for the Bully Algorithm:

- **Synchronous System:** All the processes are well-synced; they all work at the same pace.
- **Fast Communication:** Messages travel quickly between processes.
- **Timeouts:** If a process doesn't respond within a certain time, it's assumed to be not working.

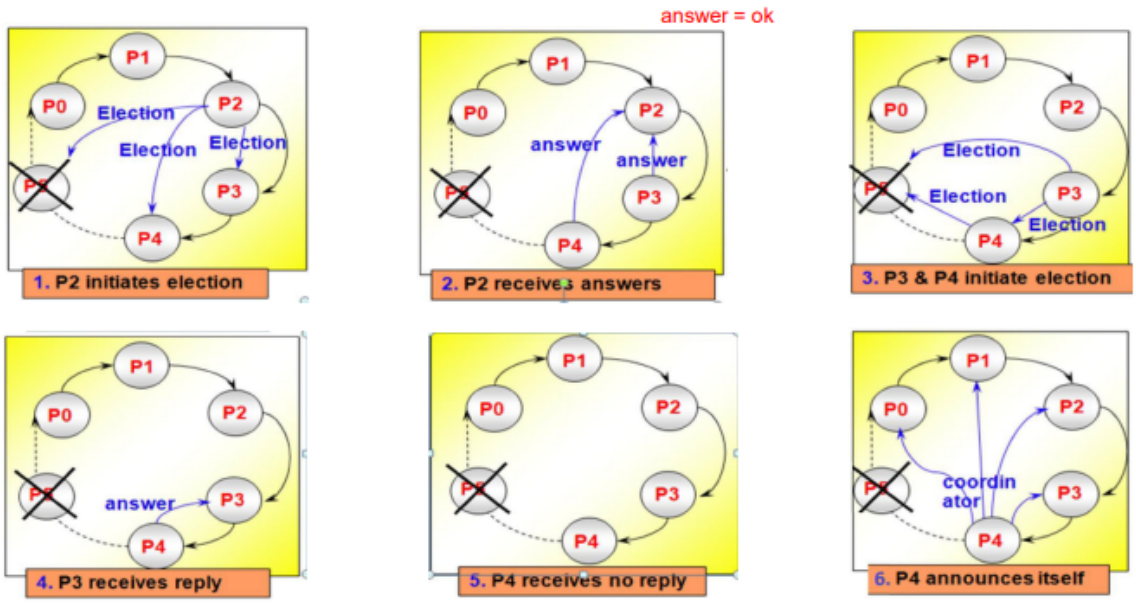
By following these rules, the Bully Algorithm helps processes in a distributed system elect a leader in an orderly way.

Assume the one-way message transmission time (T) is known.

What should be the first timeout value be, given the above assumption?

2T + (processing time) ≈ 2T

Steps in Bully Algorithm:



1. Initially, there are 5 processes, connected to each other. Process 5 is the leader with the highest number.
2. Process 5 fails.

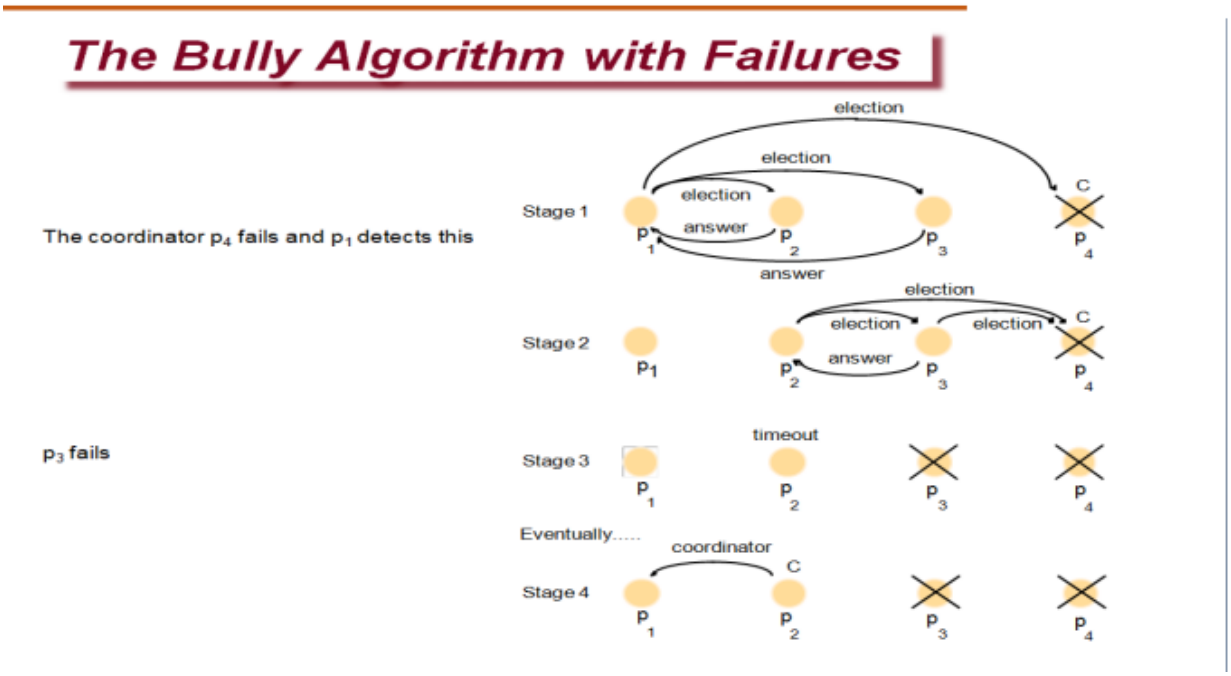
3. Process 2 notices that Process 5 is not responding. So, it starts an election, informing processes with IDs higher than itself.
4. Process 3 and Process 4 respond to Process 2, saying they will take over.
5. Process 3 starts an election and informs Process 4 and Process 5.
6. Only Process 4 responds to Process 3 and takes over the election.
7. Process 4 sends only one election message to Process 5.
8. When Process 5 doesn't respond, Process 4 declares itself the winner.

Failures During the Election Run:

There can be failures during the election run. For instance, Process 4 might fail after it sends the coordinator message, but before it sends the election message.

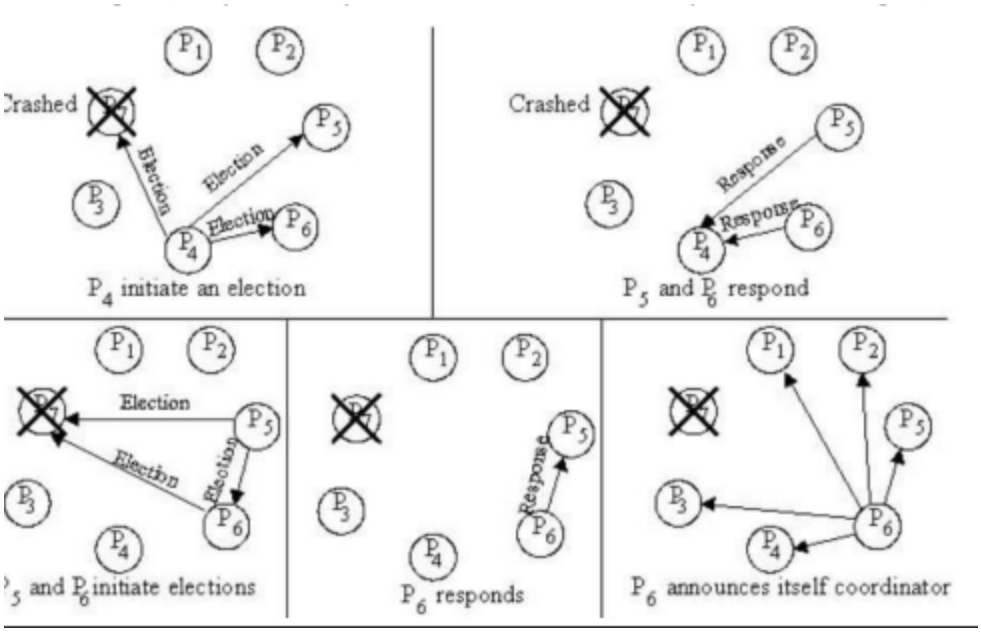
Analysis of Bully Algorithm:

- **Worst-case scenario:** When the process with the lowest ID detects the failure.
  - N-1 processes start elections, each sending messages to processes with higher IDs.
  - The message overhead is O(N^2).
  - Turnaround time is approximately 5 message transmission times if there are no failures during the run.



- **Best-case scenario:** The process with the second-highest ID notices the failure of the coordinator and elects itself.
  - N-2 coordinator messages are sent.
  - Turnaround time is one message transmission time.

Practice Problem on Bully Algorithm:



If out of 7 processes, P7 fails, and P4 detects it first, how many election messages and how many response messages are sent if all other Processes function ideally and do not fail?

Solution:

- 6 Election Messages (3 by P4, 2 by P5, and 1 by P6)
- 3 Response Messages (P5 & P6 to P4 and P6 to P5)

Resource Management and Scheduling

Introduction to Resource Management and Scheduling

Scheduling:
Scheduling involves deciding how to allocate resources of a system, such as CPU cycles, memory, secondary storage space, I/O, and network bandwidth, between users and tasks. It's a critical function that affects functionality, performance, and cost.

Resource Management and Scheduling: Motivation
Cloud resource management is complex due to:

- Multi-objective optimization: Policies and decisions are challenging.
- Lack of accurate global state information.
- Unpredictable system due to failures and attacks.
- Large fluctuating loads challenging cloud elasticity.
- Different resource management strategies for different cloud service models (IaaS, PaaS, SaaS).

Cloud Resource Management (CRM) Policies

1. **Admission control:** Prevents the system from accepting workload in violation of high-level system policies.
2. **Capacity allocation:** Allocates resources based on the need in terms of size, provisioning of different fixed, variable sizes.

3. **Optimization and Load balancing:** Distributes the workload evenly among the servers for optimization.
4. **Energy optimization:** Minimizes energy consumption through the choice of components, operating modes, etc.
5. **Quality of service (QoS) guarantees:** Ability to satisfy conditions specified by a Service Level Agreement.
6. **Locale-based:** Considers location-based legal compliance, performance, etc.
7. **Cost:** Based on perceived value or auctions.

Mechanisms for Implementing Resource Management Policies

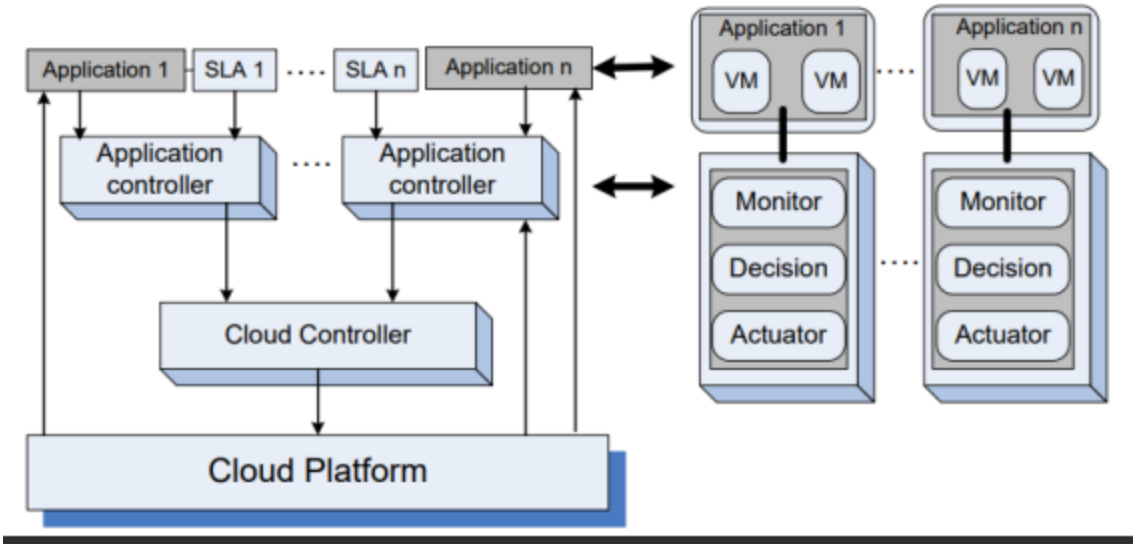
1. **Control theory:** Uses feedback such as system utilization, latencies, scheduling time, etc., to control the allocation of resources and guarantee system stability.
2. **Utility-based:** Requires a performance model and a mechanism to correlate user-level performance with cost. It considers the SLA with rewards and specifies the rewards as well as penalties associated with specific performance metrics using average response times.
3. **Machine learning:** Uses historic data to predict resource requirements.
4. **Market-oriented/economic-oriented:** Mechanisms such as combinatorial auctions are used by users to pay for resources.

Cloud Resource Management involves Tradeoffs

- Load may be concentrated on a few nodes instead of being balanced across the cluster.
- There's a significant relationship between clock frequency and system performance. It decreases at a lower rate. Similarly, voltage and energy consumption are related, where energy consumed reduces with a reduction in voltage.

Application of Control Theory for Cloud Resource Management - Illustration

The main components of a control system:



- **Inputs:** Workload, policies, capacity allocation, load balancing, energy optimization, and QoS guarantees.
- **Control system components:** Sensors used to estimate relevant performance measures and controllers that implement various policies.
- **Outputs:** Resource allocations to individual applications.

Cloud Scheduling

Scheduling is responsible for resource sharing at several levels:

- A server can be shared among several virtual machines.
- A virtual machine could support several applications.
- An application may consist of multiple threads.

A scheduling algorithm should be:

- Efficient
- Fair
- Starvation-free

Objectives of a Scheduler:

- **Batch system:** Maximize throughput and minimize turnaround time.
- **Real-time system:** Meet deadlines and be predictable.

- **Best-effort:** For batch applications and analytics.

Common algorithms for best effort applications:

- Round-robin
- First-Come-First-Serve (FCFS)
- Shortest-Job-First (SJF)
- Priority algorithms

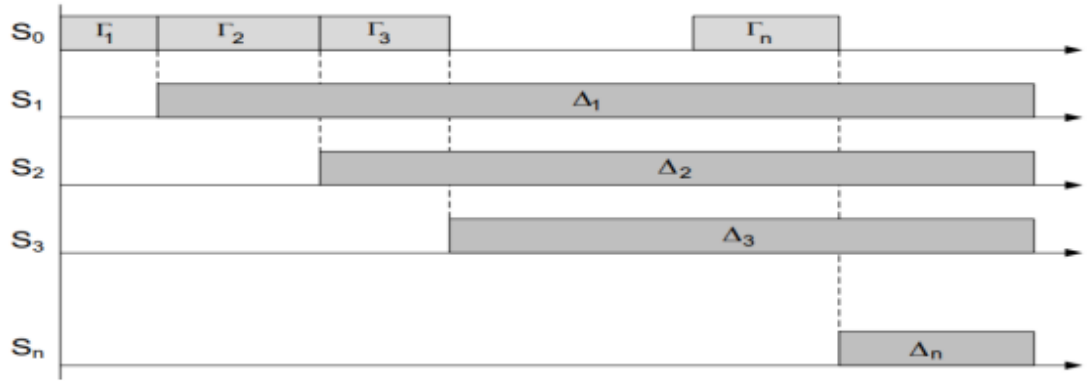
Cloud Scheduling subject to deadlines

- **Hard deadlines:**
    - If the task is not completed by the deadline, other tasks depending on it may be affected, and there are penalties.
    - Hard deadlines are strict and expressed precisely as milliseconds or seconds.
- **Soft deadlines:**
    - More of a guideline; there are generally no penalties for missing them by small amounts.
    - Soft deadlines can be missed by fractions of the units used to express them, e.g., minutes if the deadline is expressed in hours, or hours if the deadline is expressed in days.

We consider only aperiodic tasks with arbitrarily divisible workloads, e.g., energy efficiencies, affinity, etc.

Workload Partition rules

1. **Optimal Partitioning Rule (OPR):**
    - Workload is partitioned to ensure the earliest possible completion time, and all tasks are required to complete at the same time.



2. **Equal Partitioning Rule (EPR):**
    - Assigns an equal workload to individual worker nodes.

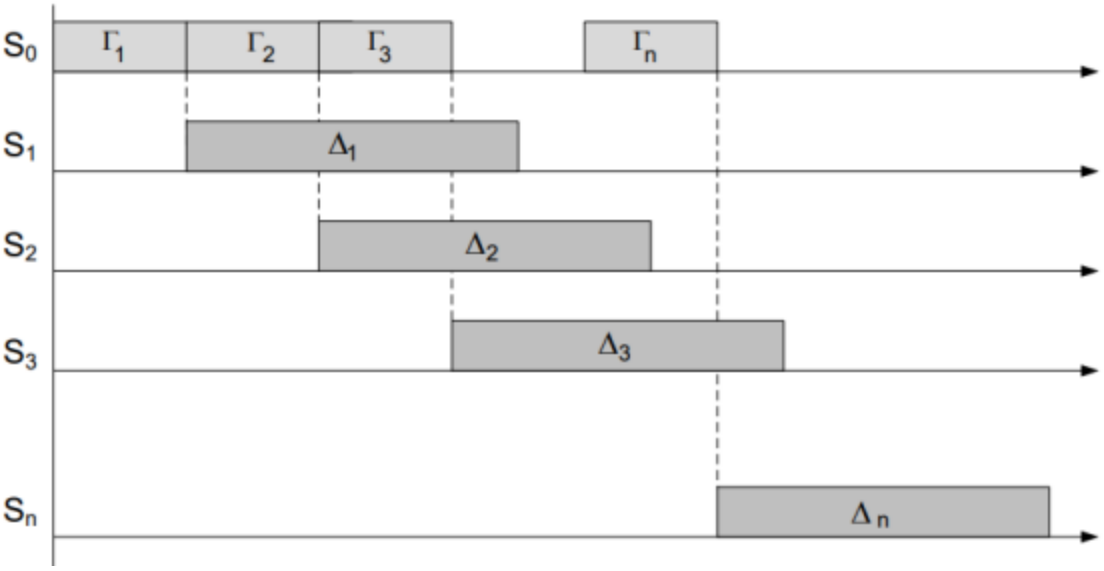
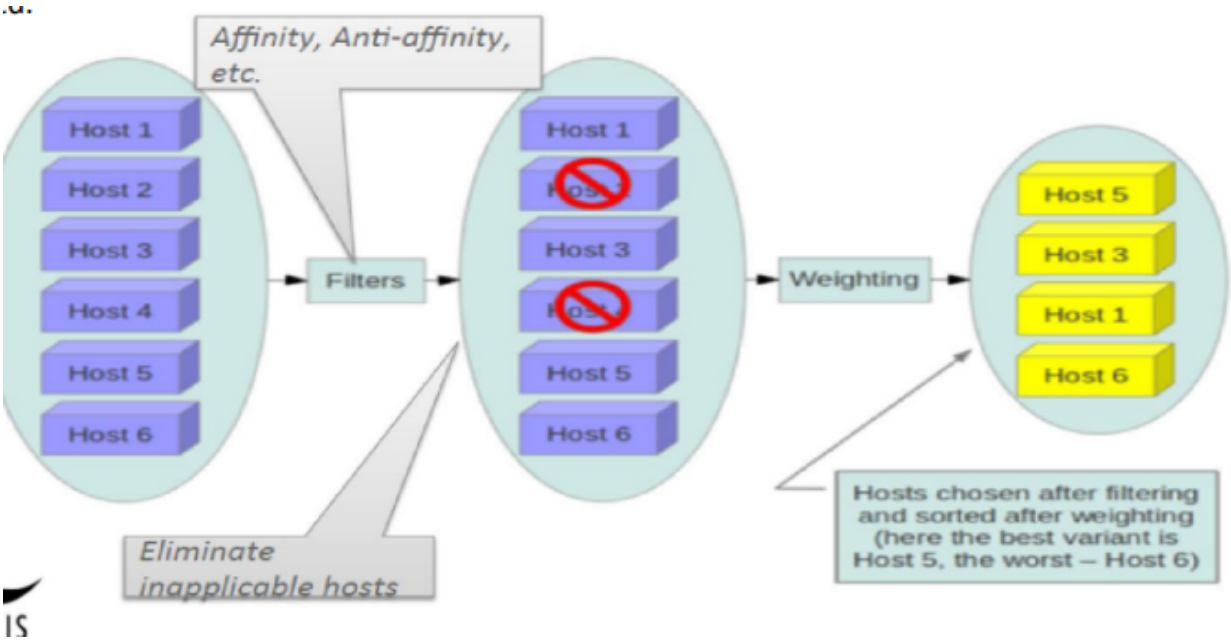


Illustration - How does Nova allocate VMs?



When a user requests a VM instance, how is it allocated?

- Compute uses the nova-scheduler service to determine how to dispatch compute requests.
- The nova-scheduler service decides on which host a VM should launch.

In the context of filters, the term "host" means a physical node that has a nova-compute service running on it. You can configure the scheduler through various options. The scheduler chooses a new host when an instance is migrated.

Distributed Locking

Definition:

- In a distributed system, truth is defined by the majority.
- A node cannot trust its own judgment.
- A distributed system cannot rely on a single node because nodes may fail anytime.
- Decisions in distributed systems are based on the votes of a minimum number of nodes, reducing dependency on any single node.

What is Distributed Locking?

- Nodes must acquire a lock before trying to access data.
- Acquiring a lock gives exclusive access to the data.
- Once the lock is acquired, the node performs the necessary operations and releases the lock.
- Distributed locking ensures mutual exclusion and prevents logical failures caused by resource contention.
- acquire, operate, release

Advantages of Distributed Locking:

1. **Ensuring Exclusive Access:**
    - Locking ensures that only one node performs work at a time among several nodes attempting to participate.
2. **Efficiency:**
    - Locking saves software from performing redundant work.
3. **Correctness:**
    - Prevents concurrent processes from corrupting data, causing data loss, or inconsistency.

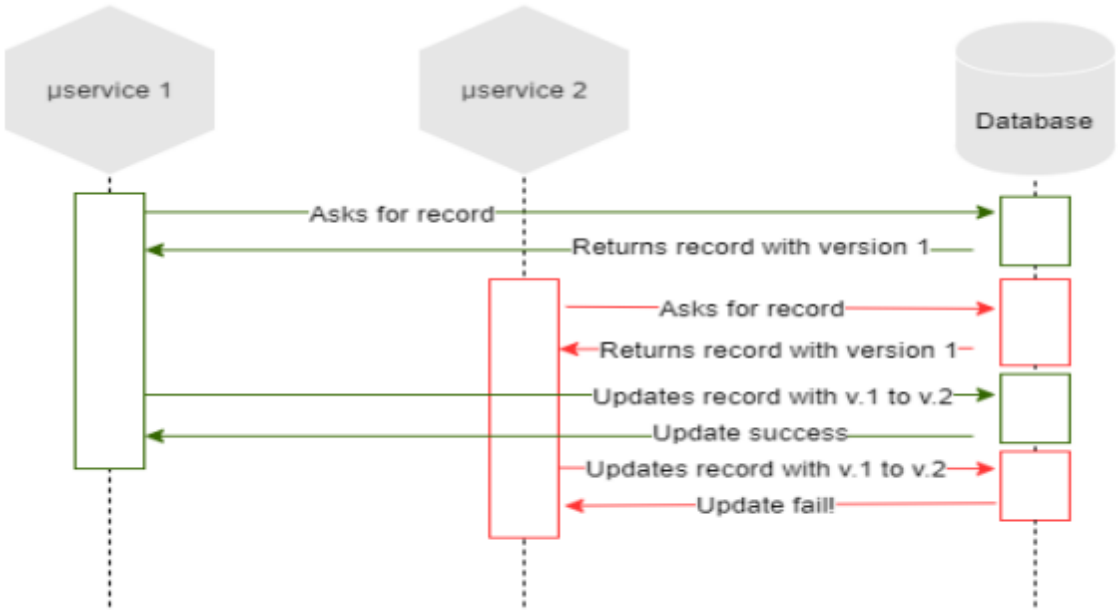
Features of Distributed Locks:

- **Mutual Exclusion:**
    - Only one client globally can hold a lock at any given moment.
- **Deadlock Free:**
    - Uses a lease-based locking mechanism where a lock is automatically released after a certain period if an exception occurs.
- **Consistency:**
    - Handles failovers triggered by external or internal errors by ensuring that the client's lock remains the same even after failovers.

Types of Distributed Locks

Optimistic Locks:

- In optimistic locking, instead of blocking, we proceed with the operation in the hope that everything will be okay.
- The lock is acquired only at the time of committing the operation.
- Before committing the operation, the system checks if the record was updated by someone else.
- Optimistic locking is suitable for scenarios where conflicts are rare.

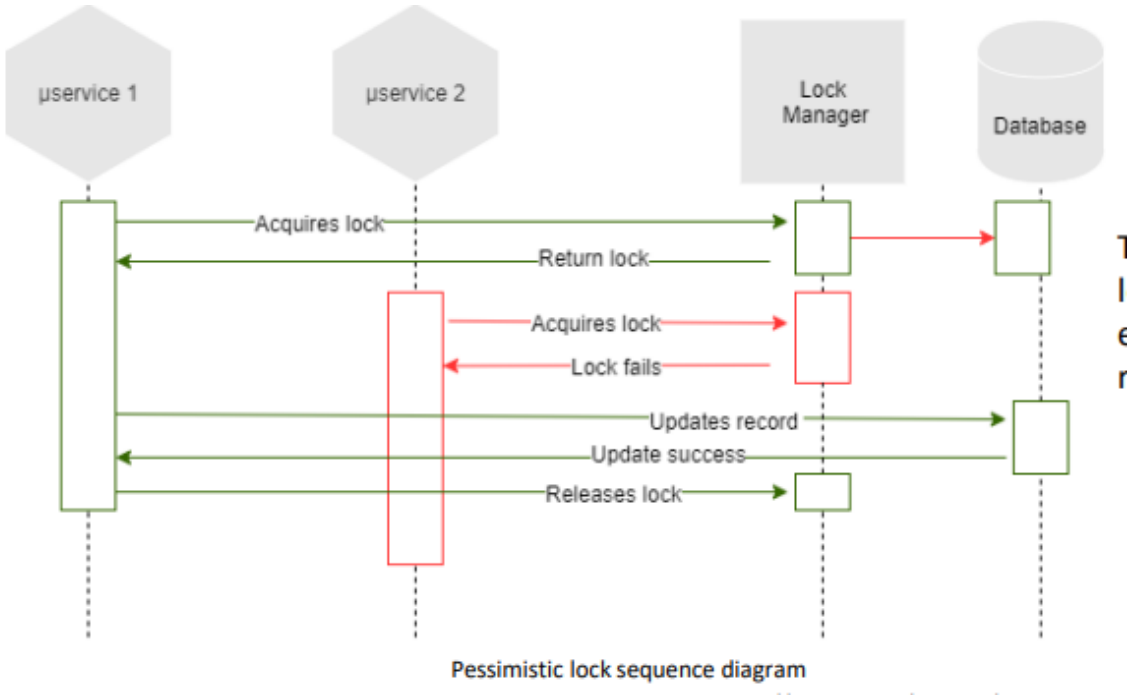


Optimistic lock sequence diagram

Check if the record was updated by someone else before the commit operation

Pessimistic Locks:

- In pessimistic locking, access to the resource is blocked before operating on it, and the lock is released at the end.
- An exclusive lock is taken to prevent any other process from modifying the record.
- Pessimistic locking is suitable for scenarios where conflicts are common and need to be avoided.



Pessimistic lock sequence diagram

IImplementing Distributed Locking

- Distributed Locking is more complex than Mutex Locking in a multi-threaded system.
- Different node or network failures can occur independently.
- Consider an application where a client needs to update a file in shared storage, for example, HDFS or S3.
- First, the client acquires the lock, then reads the file, makes changes, writes the modified file back, and finally releases the lock.
- The lock prevents two clients from performing this read-modify-write cycle concurrently, avoiding any lost updates.

```
// THIS CODE IS BROKEN
function writeData(filename, data) {
    var lock = lockService.acquireLock(filename);
    if (!lock) {
        throw 'Failed to acquire lock';
    }

    try {
        var file = storage.readFile(filename);
        var updated = updateContents(file, data);
        storage.writeFile(filename, updated);
    } finally {
        lock.release();
    }
}
```

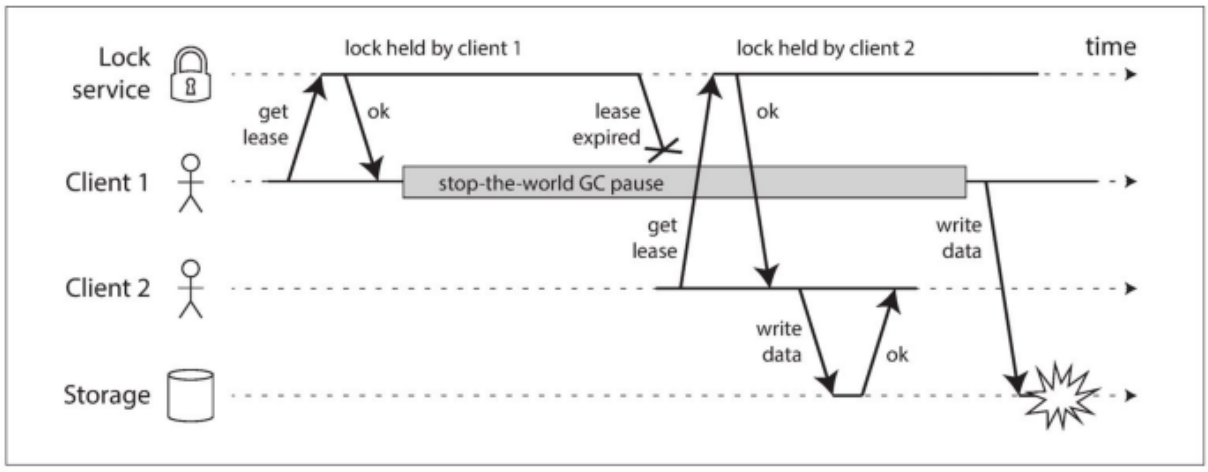
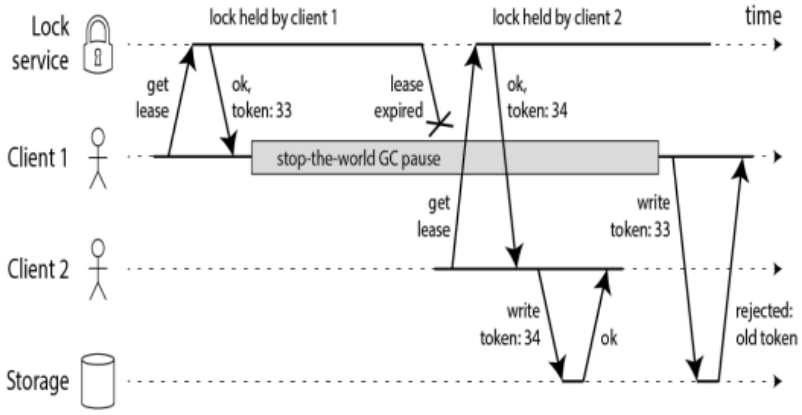


Figure 8-4. Incorrect implementation of a distributed lock: client 1 believes that it still has a valid lease, even though it has expired, and thus corrupts a file in storage.

In this example,
• the client that acquired the lock is paused for an extended period of time while holding the lock
for example because the garbage collector kicked in.
• The lock has a timeout (i.e. it is a lease), which is always a good idea (otherwise a crashed client
could end up holding a lock forever and never releasing it).
• However, if the GC pause lasts longer than the lease expiry period, and the

client doesn’t realize
that it has expired, it may go ahead and make some unsafe change.



Fencing
When using a lock or lease to protect access to some resource, such as the file storage, ensure that a node under a false belief o being “the chosen one” cannot disrupt the rest of the system.

Assume that every time the lock server grants a lock or lease, it also returns a fencing token, which
is a number that increases every time a lock is granted
• Every time a client sends a write request to the storage service, it must include its current fencing
token.
• Client 1 acquires the lease with a token of 33, but then it goes into a long pause and the lease
expires.
• Client 2 acquires the lease with a token of 34 and then sends its write request to the storage
service, including the token of 34.
• Later, client 1 comes back to life and sends its write to the storage service, including its token value
33.
• However, the storage server remembers that it has already processed a write with a higher token
number (34), and so it rejects the request with token 33.

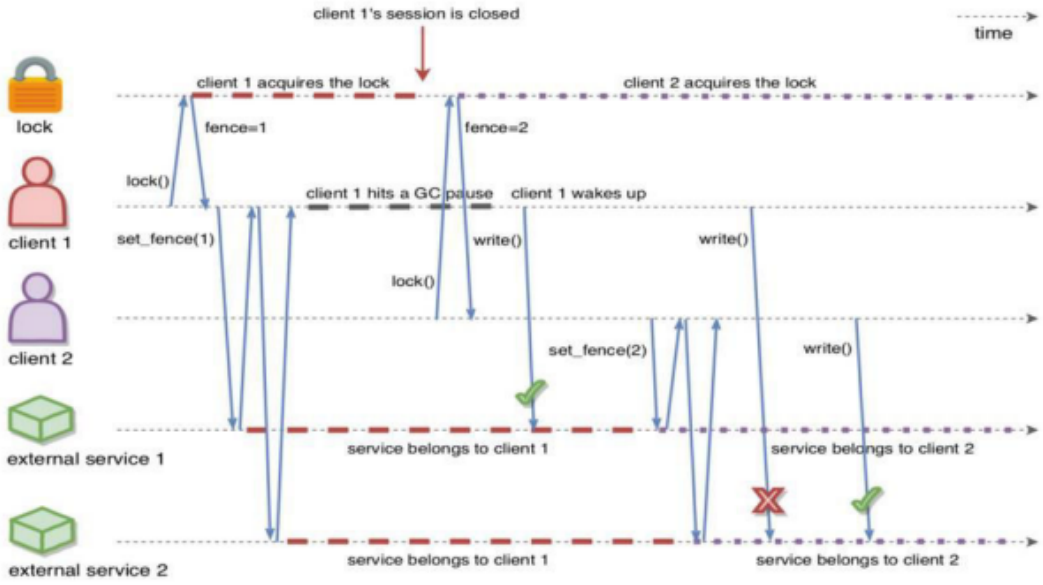


Figure 1: Using fencing tokens to fence off stale lock holders

Distributed Lock Manager

Distributed Lock Manager (DLM) is a software component that provides a means for synchronizing access to shared resources in a distributed system. Here's how it works:

- **DLM runs in every node:** Every node in the system has an identical copy of a cluster-wide lock database.
- **Synchronizing shared resources:** DLM allows software applications to synchronize their access to shared resources.
- **Generalized concept of resources:** DLM treats every resource as an entity that requires controlled shared access. This could be a file, a record, an area of shared memory, or any other resource that needs to be shared.

Some popular implementations of Distributed Lock Manager include:

1. **Google's Chubby:** Chubby is a lock service designed for loosely coupled distributed systems.
2. **Apache ZooKeeper:** ZooKeeper is an open-source software used for distributed coordination. It provides distributed locking capabilities among

other features.

3. **Redis:** Redis is an open-source, BSD licensed, advanced key-value cache and store. It can be used to implement the Redlock Algorithm for distributed lock management.

ZooKeeper

Why is ZooKeeper needed?

In large cluster environments, such as a Hadoop cluster spanning 500 or more commodity servers, centralized management of the entire cluster becomes necessary. ZooKeeper serves several purposes:

1. **Distributed Coordination:** ZooKeeper simplifies distributed coordination processes, especially in cases where message-based communication can be complex.
2. **Lock Management:** It provides support for managing locks efficiently, which can be challenging in distributed systems.
3. **Reliability and Availability:** ZooKeeper ensures reliability and availability by addressing single points of failure using an ensemble of nodes.
4. **Atomicity:** It ensures the atomicity of distributed transactions, guaranteeing that transactions either succeed or fail completely, without leaving any partial transactions.
5. **Hierarchical Namespace:** ZooKeeper helps maintain a standard hierarchical namespace similar to files and directories.

Objectives of Apache ZooKeeper:

- Open source, high-performance coordination service for distributed applications across many hosts.
- Automation of cluster management, allowing developers to focus on building software features rather than worrying about distributed nature.
- Provides a centralized coordination service for maintaining configuration information, performing distributed locks and synchronization, and enabling group services.

- Uses a shared hierarchical namespace of data registers (znodes) to coordinate distributed processes.
- Exposes common services through a simple interface: Naming, configuration management, locks and synchronization, and group services. Developers don't need to write these services from scratch.

How Apache ZooKeeper Functions:

- ZooKeeper provides an infrastructure for cross-node synchronization by maintaining status type information in memory on ZooKeeper servers.
- Applications can create znodes, which are files that persist in memory on ZooKeeper servers. These znodes can be updated by any node in the cluster, and any node in the cluster can register to be notified of changes to that znode.
- Applications can synchronize their tasks across the distributed cluster by updating their status in a ZooKeeper znode.
- ZooKeeper servers keep a copy of the state of the entire system and persist this information in local log files. Large Hadoop clusters are supported by multiple ZooKeeper servers, with a master server synchronizing the top-level servers.
- This cluster-wide status centralization service is critical for management and serialization tasks across a large distributed set of servers.

Apache ZooKeeper Features:

1. **Updating Node's Status:** Updates every node, allowing it to store updated information about each node across the cluster.
2. **Managing the Cluster:** Manages the cluster in real-time, maintaining the status of each node and reducing chances for errors and ambiguity.
3. **Naming Service:** Attaches a unique identification to every node, similar to DNA, helping identify each node.
4. **Automatic Failure Recovery:** Locks the data while modifying, allowing the cluster to recover automatically if a failure occurs in the database.

Apache ZooKeeper Architecture

Server:

- When a client connects, the server sends an acknowledgement.
- A leader is elected at startup.
- All servers store a copy of the data tree in memory.

Client:

- Every client sends a message to the server at regular intervals to indicate that it's alive.
- If there is no response from the connected server, the client automatically redirects the message to another server.

Leader:

- One of the servers is designated as the Leader.
- The Leader provides information to the clients and acknowledges that the server is alive.
- It performs automatic recovery if any of the connected nodes fail.

Follower:

- A server node that follows the leader's instructions is called a Follower.
- Client read requests are handled by the correspondingly connected ZooKeeper server.
- Client write requests are handled by the ZooKeeper leader.

Ensemble/Cluster:

- A group of ZooKeeper servers is called an ensemble or a cluster.
- Running ZooKeeper in cluster mode optimizes the system.

ZooKeeper WebUI:

- Provides a web user interface for working with ZooKeeper.
- Allows for fast and effective communication with the ZooKeeper application, rather than using the command line.

Apache ZooKeeper Data Model

Zookeeper Data Model:

- Follows a hierarchical namespace similar to a file system.
- Each node in the system is created with a "znode."
- Each znode has data (configurations, status, counters, parameters, location information) and may or may not have children.
- Keeps metadata information.
- Does not store big datasets.
- ZNode paths are canonical, slash-separated, and absolute, without any relative references.

Types of Znodes:

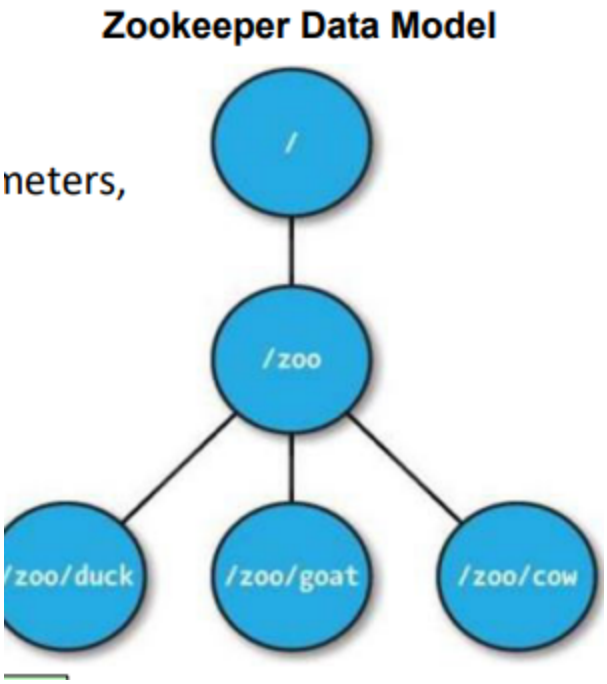
1. **Persistent znodes:**
    - Permanent and must be deleted explicitly by the client.
    - Persist even after the session that created the znode is terminated.
2. **Ephemeral znodes:**
    - Temporary and exist as long as the session that created the znode is active.
    - Deleted automatically when the client session creating them ends.
    - Used to detect the termination of a client.
    - Alerts, known as watches, can be set up to detect the deletion of the znode.
    - These nodes can't have children.

Sequence Nodes (Unique Naming):

- Append a monotonically increasing counter to the end of the path.
- Applies to both persistent and ephemeral nodes.
- Once a znode is created, its type cannot be changed later.

Apache ZooKeeper: Namespace and Data Organization

- ZooKeeper provides a namespace similar to a standard shared file system but is more like a distributed, consistent shared memory organized hierarchically like a file system.
- Every znode is identified by a name, which is a sequence of path elements separated by a slash ("/").
- Every znode has a parent except the root ("/").
- A znode cannot be deleted if it has any children.

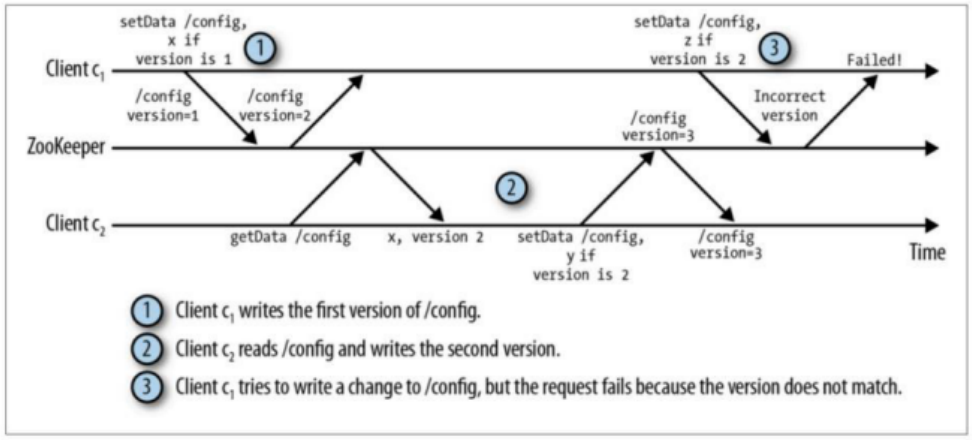


- Each znode can have data associated with it, limited to kilobytes.
- ZooKeeper is designed to store coordination data such as status information, configuration, location information, etc.
- ZooKeeper maintains an in-memory image of the data tree, replicated on all servers executing the ZooKeeper service.
- Only transaction logs and snapshots are stored in a persistent store, ensuring high throughput.
- Transaction logs and snapshots are stored in a persistent store, ensuring high throughput.

- Changes made to znodes during application execution are appended to the transaction logs.
- Each client connects to a single ZooKeeper server and maintains a TCP connection.
- The TCP connection is used for sending requests, receiving responses, getting watch events, and sending heartbeats.
- If the TCP connection to the server breaks, the client will connect to an alternate server.

Implementation:

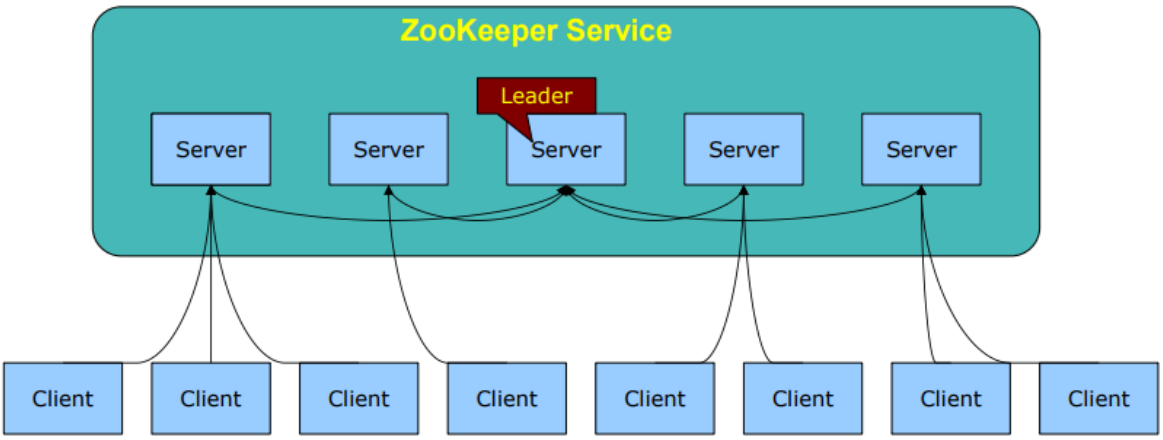
- Each znode maintains a stat structure that includes version numbers for data changes, ACL changes, and timestamps.
- When a client performs an update or delete operation, it must supply the version of the data of the znode it is changing.
- If the version supplied by the client doesn't match the actual version of the data, the update will fail.
- All updates made by ZooKeeper are totally ordered and stamped with a unique sequence called zxid (ZooKeeper Transaction Id).



Each Znode has version number, is incremented every time its data changes
- *setData* and *delete* take version as input, operation succeeds only if client's version is equal to server's one

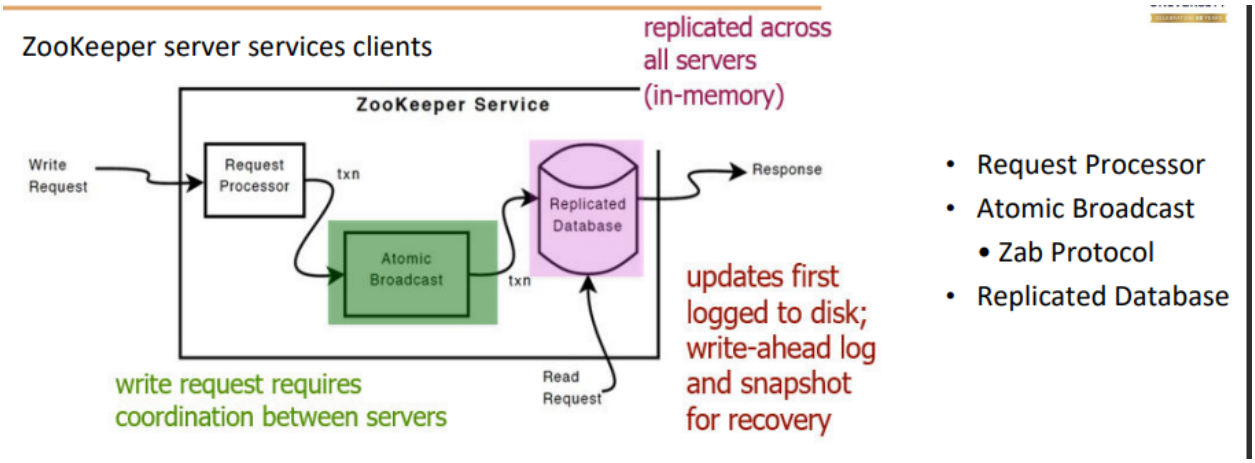
ZooKeeper Reads/Writes/Watches:

- Reads: Occur between the client and a single server.
- Writes: Sent between servers first, then consensus is reached before responding back to the client.
- Watches: Also occur between the client and a single server.
- A watch on a znode essentially means "monitoring it".



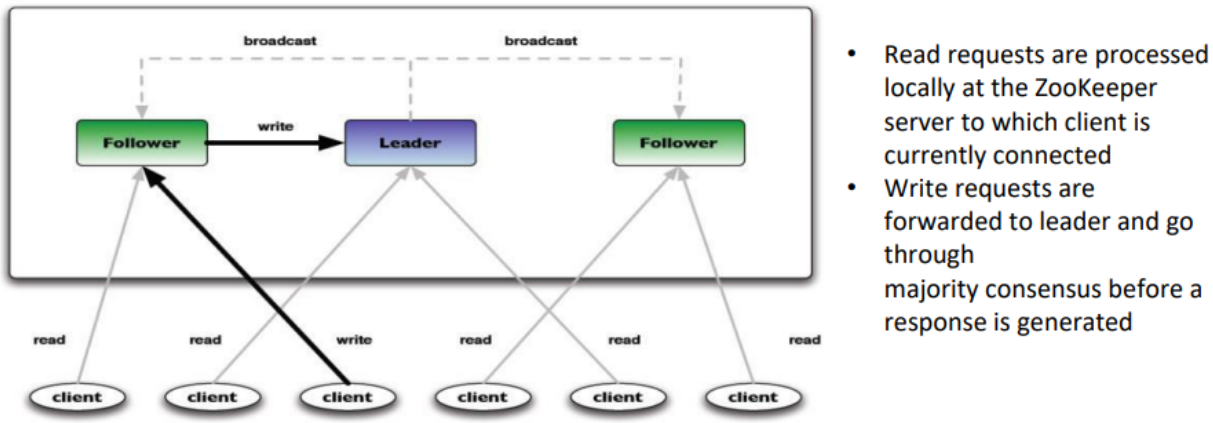
Implementation of ZooKeeper:

- **Request Processor:** Handles incoming client requests.
- **Atomic Broadcast:** Implemented using the Zab Protocol.
- **Zab Protocol:** Ensures that write requests are replicated across the ZooKeeper ensemble.
- **Replicated Database:** Each ZooKeeper server maintains a replica of the database.



ZooKeeper Reads/Writes:

- Read requests are processed locally at the ZooKeeper server to which the client is currently connected.
- Write requests are forwarded to the leader and go through a majority consensus process before a response is generated.



Operation	Type
create	Write
delete	Write
exists	Read
getChildren	Read
getData	Read
setData	Write
getACL	Read
setACL	Write
sync	Read

ZooKeeper Operations

ZooKeeper provides a simple programming interface with the following operations:

1. **create:** Creates a node at a location in the tree.
2. **delete:** Deletes a node.
3. **exists:** Tests if a node exists at a location.
4. **get data:** Reads the data from a node.
5. **set data:** Writes data to a node.
6. **get children:** Retrieves a list of children of a node.
7. **sync:** Waits for data to be propagated.

Advantages of ZooKeeper

1. **Simple distributed coordination process:** Provides a simple programming interface for coordination tasks.

2. **Synchronization:** Enables mutual exclusion and cooperation between server processes.
3. **Ordered Messages:** Ensures that messages are delivered in the order they were sent.
4. **Serialization:** Encodes data according to specific rules, ensuring consistent application execution.
5. **Reliability:** Provides a highly reliable coordination service.
6. **Atomicity:** Ensures that data transfer either succeeds or fails completely, with no partial transactions.

Disadvantages of ZooKeeper

1. **Lack of built-in service discovery:** Other solutions like Consul offer features like service discovery with health checks, which ZooKeeper lacks.
2. **Ephemeral node lifecycle tied to TCP connection:** The lifecycle of ephemeral nodes is tied to the TCP connection, which may lead to issues if the process on the other side is not functioning properly.
3. **Complexity:** ZooKeeper can be complex to set up and manage.

ZooKeeper Recipes: Why should I use ZooKeeper?

1. **Naming service:** Provides a simple interface to track the status of servers or services by name.
2. **Distributed Locks:** Enables implementing distributed mutexes for serialized access to shared resources.
3. **Data Synchronization:** Helps in synchronizing access to shared resources for ensuring consistency.
4. **Cluster/Configuration management:** Centrally stores and manages the configuration of a distributed system.
5. **Leader election:** Provides support for automatic fail-over strategy in case of node failures.
6. **Message queue:** Facilitates inter-node communication through a queue system implemented using znodes.

Companies using ZooKeeper:

- Yahoo!
- Zynga
- Rackspace
- LinkedIn
- Netflix

Projects using ZooKeeper:

- Apache MapReduce (YARN)
- Apache HBase
- Apache Kafka
- Apache Storm
- Neo4j

Reverse Proxy

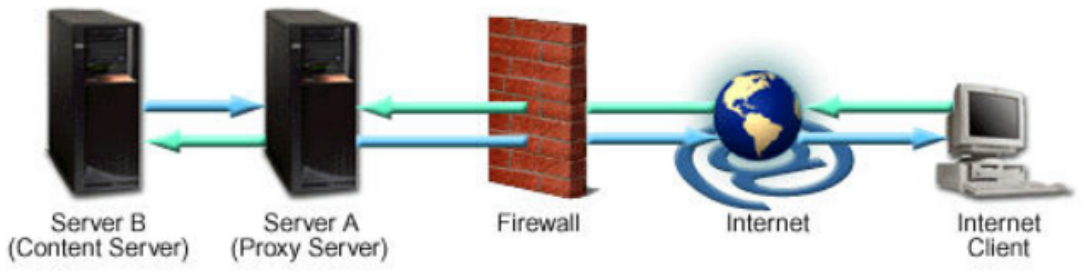
What is a reverse proxy?

A reverse proxy is a server that sits in front of web servers and forwards client (e.g., web browser) requests to those web servers.

How it works:

- The reverse proxy server first checks to ensure that a request is valid. If a request is not valid or not allowed (blocked by the proxy), it will not continue to process the request, resulting in the client receiving an error or a redirect.
- If a request is valid, a reverse proxy may check if the requested information is cached. If it is, the reverse proxy serves the cached information.
- If it is not cached, the reverse proxy requests the information from the content server and serves it to the requesting client. It also caches the information for future requests.

Example:



- An Internet client initiates a request to Server A (Proxy Server), which, unknown to the client, is actually a reverse proxy server.
- The request is allowed to pass through the firewall and is valid but is not cached on Server A.
- The reverse proxy (Server A) requests the information from Server B (Content Server), which has the information the Internet client is requesting.
- The information is served to the reverse proxy, where it is cached, and relayed through the firewall to the client.
- Future requests for the same information will be fulfilled by the cache, lessening network traffic and load on the content server. (Proxy caching is optional and not necessary for the proxy to function on your HTTP Server).

Benefits of reverse proxy

1. **Improved online security:**
    - Reverse proxies play a key role in building a zero trust architecture for organizations, securing sensitive business data and systems.
    - They only forward requests that your organization wants to serve, ensuring that no information about your backend servers is visible outside your internal network.
    - Reverse proxies protect backend servers from being directly accessed by malicious clients, thus safeguarding them from distributed denial-of-service (DDoS) attacks.
2. **Increased scalability and flexibility:**
    - Reverse proxies increase scalability and flexibility, especially in a load-balanced environment where the number of servers can be scaled up and

down based on fluctuations in traffic volume.

- Clients see only the reverse proxy's IP address, allowing for changes in the backend infrastructure configuration without affecting the clients.
- Load balancing techniques distribute traffic over one or multiple servers, improving overall performance and ensuring applications no longer have a single point of failure.

3. **Web acceleration:**
    - Reverse proxies help with web acceleration by reducing the time taken to generate a response and return it to the client.
4. **Identity branding:**
    - Businesses can use reverse proxies to conceal the fact that they are sending site visitors to a different URL for payment, improving identity branding.
5. **Caching commonly-requested data:**
    - Reverse proxies can cache commonly-requested data, such as images and videos, relieving pressure on internal services, speeding up performance, and improving user experience, especially for sites with dynamic content.

What is a forward proxy?

A forward proxy, often called a proxy, proxy server, or web proxy, is a server that sits in front of a group of client machines. When those computers make requests to sites and services on the Internet, the proxy server intercepts those requests and then communicates with web servers on behalf of those clients, acting as a middleman.

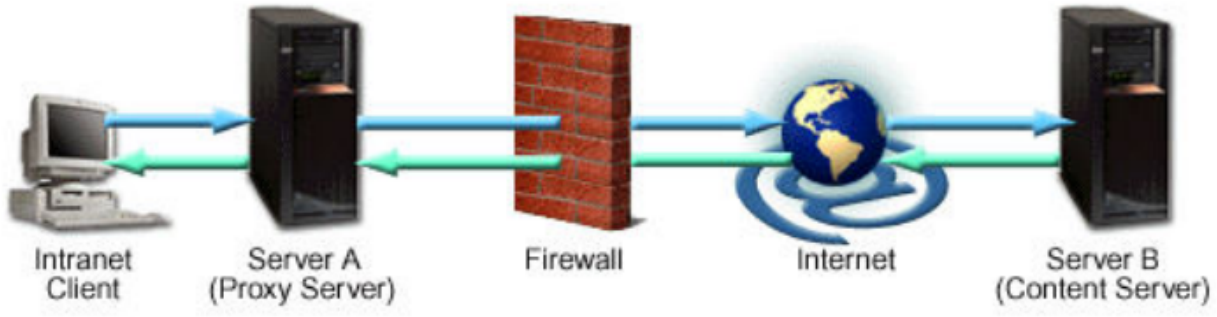
Reasons for using a forward proxy:

1. To avoid state or institutional browsing restrictions.
2. To block access to certain content.
3. To protect their identity online.

How it works:

- A forward proxy server first checks to make sure a request is valid. If a request is not valid or not allowed (blocked by the proxy), it rejects the request, resulting in the client receiving an error or a redirect.
- If a request is valid, a forward proxy may check if the requested information is cached. If it is, the forward proxy serves the cached information.
- If the requested information is not cached, the request is sent through a firewall to an actual content server, which serves the information to the forward proxy. The proxy relays this information to the client and may also cache it for future requests.

Example:
An intranet client initiates a request that is valid but not cached on Server A (Proxy Server). The request is sent through the firewall to the Internet server, Server B (Content Server), which has the information the client is requesting. The information is sent back through the firewall, where it is cached on Server A and served to the client. Future requests for the same information will be fulfilled by the cache, reducing network traffic. (Proxy caching is optional and not necessary for a forward proxy to function on your HTTP Server).



Benefits of forward proxy:

1. **Avoid state or institutional browsing restrictions:**

   Some governments, schools, and organizations use firewalls to limit access to the Internet. A forward proxy allows users to bypass these restrictions by connecting to the proxy instead of directly to the sites they want to visit.

2. **Block access to certain content:**

Conversely, proxies can be configured to block a group of users from accessing specific sites. For example, a school network might connect to the web through a proxy with content filtering rules, preventing access to social media sites like Facebook.

3. **Protect identity online:**

   Forward proxies can provide increased anonymity for regular Internet users. In regions where governments impose severe consequences on political dissidents, individuals may face fines or imprisonment for expressing dissent online. By using a forward proxy to connect to websites where they post politically sensitive comments, dissidents can make it harder to trace their IP addresses. Only the IP address of the proxy server will be visible, providing an additional layer of protection.

S.No.	FORWARD PROXY	REVERSE PROXY
1	Forward proxy connection initiates from inside secured zone and destined to outside unsecured global network.	Reverse proxy connection comes from outside global network and destined to inside secured network.
2	Forward proxy are not used for Application Delivery.	Reverse proxy are built for Application Delivery.
3	Forward proxy are good for content filtering, natting, Email Security etc.	Reverse Proxy are used for Load Balancing (TCP Multiplexing), Content Switching, Authentication and application firewall.
4	Forward proxy restrict the internal user from accessing the user filtered/restricted site.	Reverse proxy restrict the outside user/client to have direct access to internal/private networks.

https://ipwithease.com

Forward Proxies are good for:

- Content Filtering
- Email security
- NAT (Network Address Translation)

- Compliance Reporting

Reverse Proxies are good for:

- Application Delivery including:
  - Load Balancing (TCP Multiplexing)
  - SSL Offload/Acceleration (SSL Multiplexing)
  - Caching
  - Compression
  - Content Switching/Redirection
  - Application Firewall
  - Authentication
  - Single Sign-On

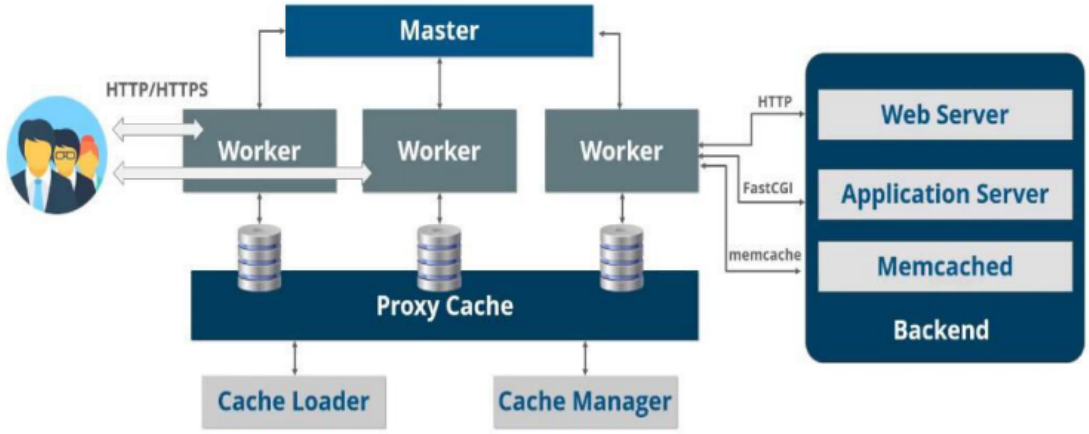
Nginx

Nginx is an open-source web server that offers various capabilities, including reverse proxying, caching, load balancing, media streaming, and more. Originally designed for maximum performance and stability, Nginx can function not only as an HTTP server but also as a proxy server for email (IMAP, POP3, and SMTP) and as a reverse proxy and load balancer for HTTP/2, TCP, and UDP protocols.

Nginx as High Requests Handling Architecture

Nginx utilizes an event-driven architecture, handling requests asynchronously. It employs a non-blocking event-driven connection handling algorithm, enabling it to process thousands of connections (requests) within a single processing thread. Nginx can handle over 10,000 simultaneous connections with low CPU and memory resources under heavy request loads.

Nginx Architecture



Nginx follows a master-slave architecture, supporting an event-driven, asynchronous, and non-blocking model.

- **Master:** The master allocates jobs for the workers based on client requests. It doesn't wait for responses from the workers but instead looks for the next request from the client. Once the response is received from the workers, the master sends it to the client.
- **Workers:** Workers are the slaves in the Nginx architecture and follow the master's directives. Each worker can handle over 1000 requests simultaneously in a single-threaded manner. After processing, the response is sent to the master. The single-threaded approach saves RAM and ROM size by working on the same memory space instead of different memory spaces.
- **Cache:** Nginx cache stores pages on the first request, rendering subsequent page requests faster by retrieving them from cache memory instead of the server.

Benefits of Nginx

- **Load balancing:** Nginx evenly distributes client requests to multiple upstream servers, enhancing performance and providing redundancy in case of server failure. This ensures the application is always available to serve client requests, improving SLAs.
- **Security:** Nginx provides security to backend servers in private networks by hiding their identity. Backend servers are unknown to clients making requests,

and Nginx serves as a single point of access to multiple backend servers regardless of the backend network topology.

- **Caching:** Nginx can directly serve static content like images, videos, etc., improving performance by reducing the load on the backend server.
- **Logging:** Nginx offers centralized logging for backend server requests and responses passing through it, providing a single place for auditing and troubleshooting.
- **TLS/SSL support:** Nginx enables secure communication between client and server using TLS/SSL connections, ensuring user data remains secure and encrypted during transfer over the wire using HTTPS connections.
- **Protocol Support:** Nginx supports various protocols, including HTTP, HTTPS, HTTP/1.1, HTTP/2, and gRPC, along with both IPv4 and IPv6 internet protocols.

Scaling Computation- Hybrid Cloud and Cloud Bursting

Cloud scalability in cloud computing refers to the ability to increase or decrease IT resources as needed to meet changing demand. This includes data storage capacity, processing power, and networking, all of which can be scaled using existing cloud computing architecture.

Benefits of Cloud Scalability

1. **Cost Savings:** Cloud scalability allows businesses to avoid the upfront costs of purchasing expensive equipment that could become outdated in a few years. Through cloud providers, they pay for only what they use, minimizing waste.
2. **Disaster Recovery:** With scalable cloud computing, businesses can reduce disaster recovery costs by eliminating the need for building and maintaining secondary data centers.
3. **Convenience:** IT administrators can easily add more virtual machines (VMs) with just a few clicks, and these resources are available without delay and can be customized to the exact needs of an organization. This saves precious time

for IT staff, allowing them to focus on other tasks instead of spending hours and days setting up physical hardware.

4. **Flexibility and Speed:** Cloud scalability allows IT to respond quickly as business needs change and grow, including unexpected spikes in demand. Even smaller businesses have access to high-powered resources that were once cost-prohibitive. Companies are no longer tied down by obsolete equipment; they can update systems and increase power and storage with ease.

Cloud Scalability vs. Cloud Elasticity

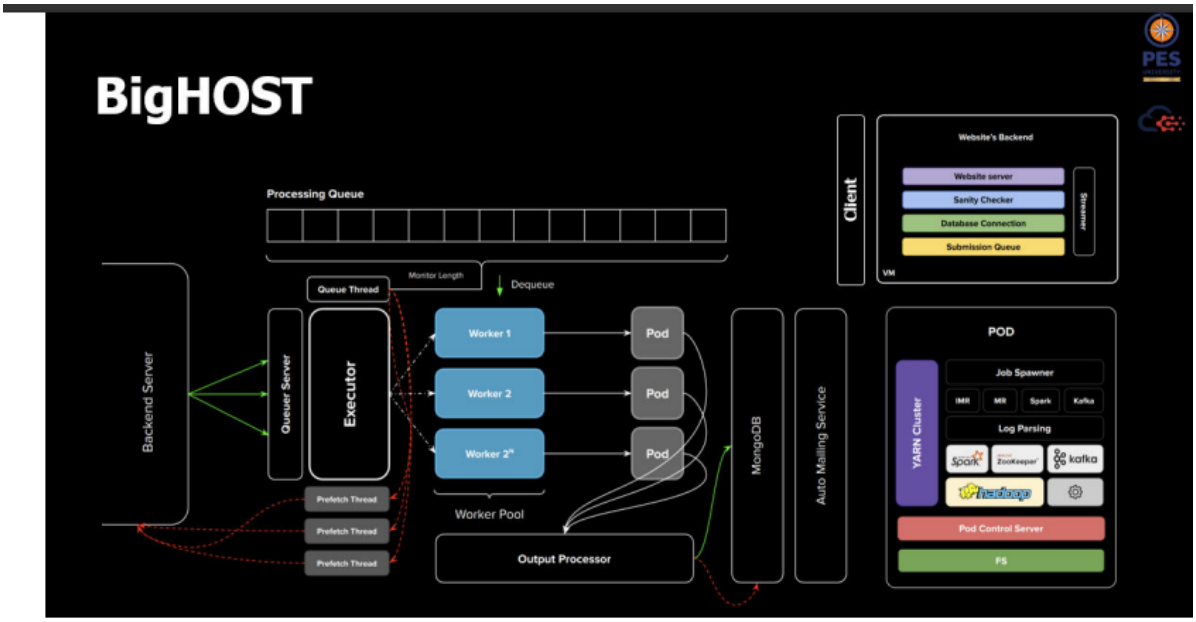
- **Elasticity:** Refers to a system's ability to grow or shrink dynamically in response to changing workload demands, such as a sudden spike in web traffic. An elastic system automatically adapts to match resources with demand as closely as possible in real time.
- **Scalability:** Refers to a system's ability to increase workload with existing hardware resources. A scalable solution enables stable, longer-term growth in a pre-planned manner, while an elastic solution addresses more immediate, variable shifts in demand.

Both elasticity and scalability are important features for a system, but the priority of one over the other depends in part on whether a business has predictable or highly variable workloads.

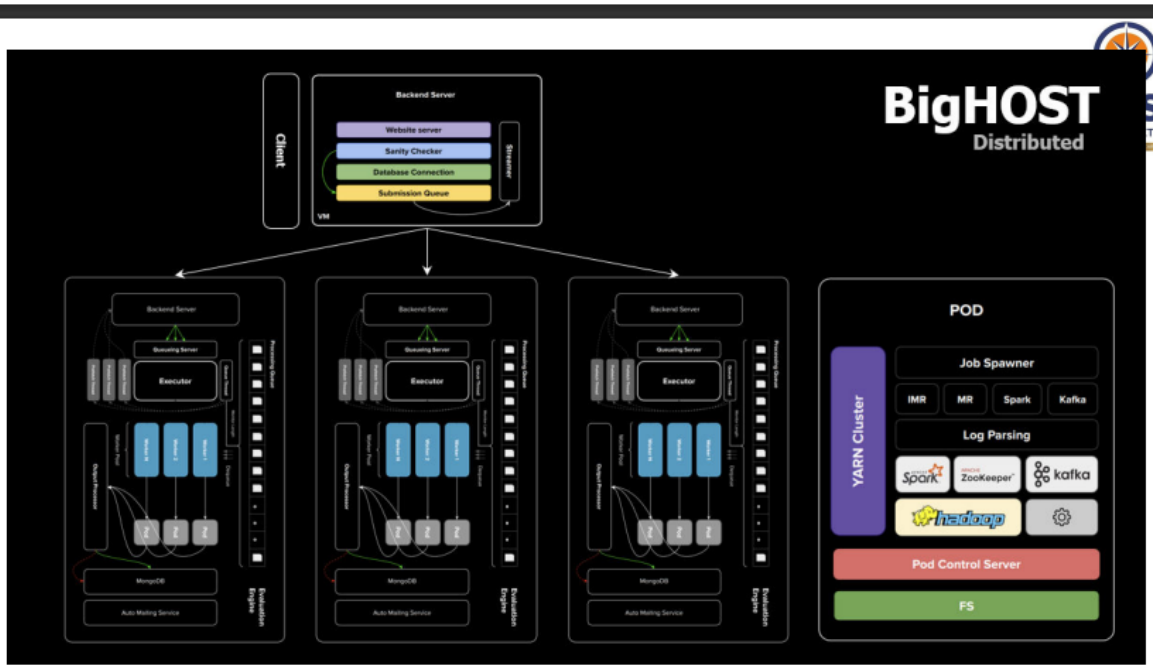
Cloud Scaling Strategies

1. **Vertical Scaling:**
    - **Definition:** Vertical scaling refers to adding more CPU, memory, or I/O resources to an existing server or replacing one server with a more powerful server.
    - **Implementation:** Cloud providers like Amazon Web Services (AWS) and Microsoft Azure allow vertical scaling by changing instance sizes. In a data center, it can be achieved by purchasing a new, more powerful appliance and discarding the old one.
    - **Example:** Scaling up an EC2 instance or RDS database to a larger size on AWS or Azure.

2. **Horizontal Scaling:**
    - **Definition:** Horizontal scaling refers to provisioning additional servers to meet your needs, often splitting workloads between servers to limit the number of requests any individual server is getting.
    - **Implementation:** In a cloud-based environment, horizontal scaling means adding additional instances instead of moving to a larger instance size.
    - **Example:** Adding more EC2 instances to distribute the workload instead of increasing the size of existing instances.



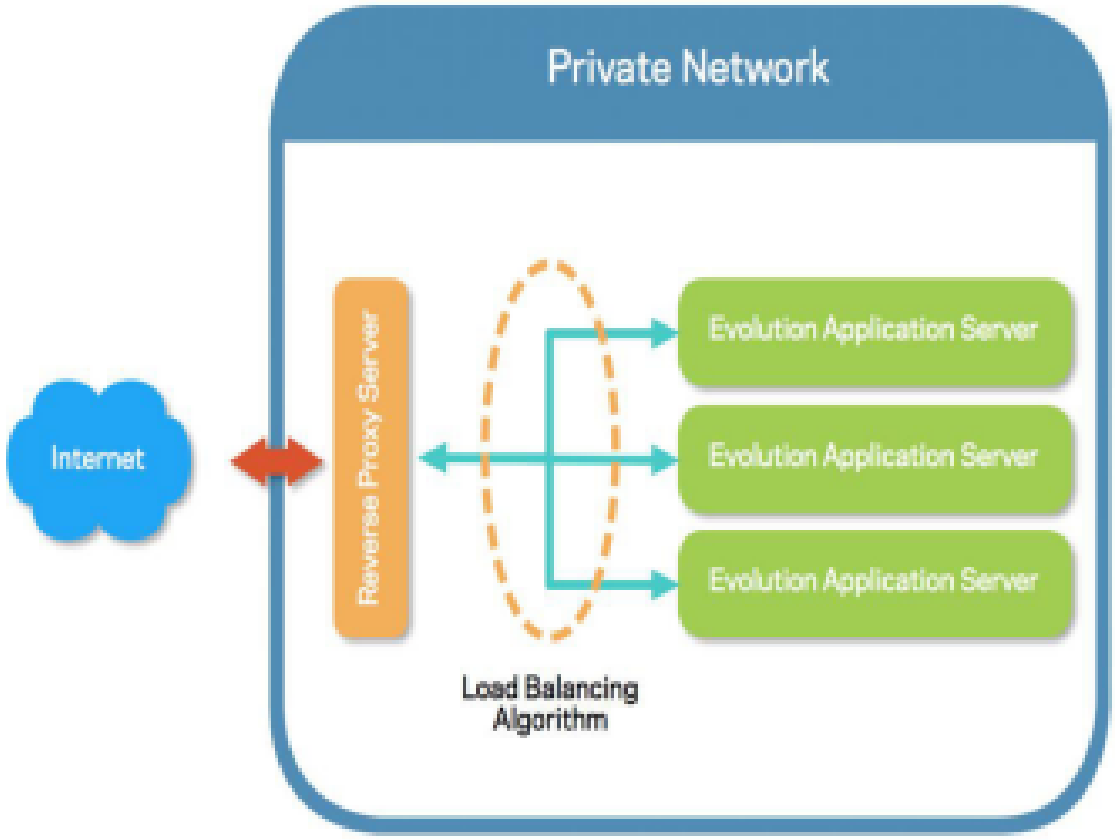
Ramesha, V., Shankar, S., Thalanki, S., Kurpad, S. and Auradkar, P., 2023, May. BigHOST: Automatic Grading System for Big Data Assignments. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)* (pp. 256-262). IEEE.



Scalability through Reverse Proxy

- **Role of Reverse Proxy in Scalability:**
  - A reverse proxy server can handle many different roles depending on the size and complexity of your network.

- In a cloud-based environment, a reverse proxy can act as a load balancer, distributing incoming traffic among multiple application servers.



- The reverse proxy presents a single server to the outside world, but it forwards each request to an application server on the private network.
- The load balancer (reverse proxy) decides which server should receive the request, improving the scalability and availability of the application.

Hybrid Cloud and Cloud Bursting

- **Definition of Hybrid Cloud:**
  - Hybrid cloud is a combination of private and public clouds that enables the expansion of local infrastructure to commercial infrastructure on an as-needed basis.

- **Benefits:**
  - Allows organizations to leverage existing in-house infrastructure and seamlessly include or supplement additional resources from the public cloud based on demand.
  - Ensures good utilization of existing infrastructure, making it a cost-effective solution.
  - Provides agility, allowing organizations to adapt and change direction quickly, which is essential for digital businesses.

Hybrid Cloud Benefits

1. **Dynamic Workloads:**
   - Use a scalable public cloud for dynamic workloads while leaving less volatile or more sensitive workloads to a private cloud or on-premises data center.
   - Example: Utilize a public cloud for fluctuating workloads, such as web traffic spikes, and a private cloud for mission-critical applications.
2. **Data Separation:**
   - Store sensitive financial or customer information on a private cloud while running the rest of your enterprise applications on a public cloud.
   - Example: Keep sensitive data behind your firewall in a private cloud while using a public cloud for less sensitive applications like email and collaboration tools.
3. **Big Data Processing:**
   - Use a public cloud for big data analytics to benefit from its scalability while ensuring data security and keeping sensitive big data behind your firewall in a private cloud.
   - Example: Analyze large datasets using scalable public cloud resources while securely storing sensitive data in a private cloud.
4. **Incremental Cloud Adoption:**
   - Move to the cloud incrementally at your own pace by putting some workloads on a public cloud or a small-scale private cloud and expanding

your cloud presence as needed.

- Example: Start with non-critical workloads on a public cloud to test the waters and gradually move more workloads to the cloud as you gain confidence.

Hybrid Cloud Scenarios

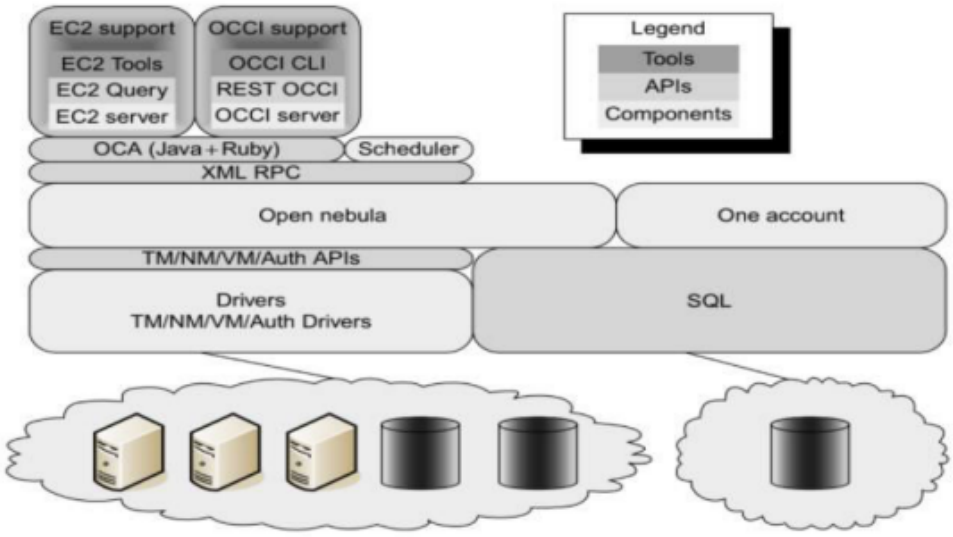
1. **Temporary Processing Capacity Needs:**
   - Allocate public cloud resources for short-term projects at a lower cost than using your own data center's IT infrastructure.
   - Example: Use public cloud resources to handle temporary spikes in demand, such as during seasonal sales or marketing campaigns.
2. **Flexibility for the Future:**
   - Match your actual data management requirements to the best-suited resources, whether public cloud, private cloud, or on-premises, using a hybrid cloud approach.
   - Example: Choose the most appropriate infrastructure for each workload, ensuring flexibility and scalability as your needs evolve.
3. **Best of Both Worlds:**
   - Enjoy the advantages of both public and private cloud solutions simultaneously, rather than limiting your options to one or the other.
   - Example: Benefit from the scalability and flexibility of public cloud resources while maintaining control and security over sensitive data in a private cloud.

Cloud Bursting

- **Definition:**
  - Cloud bursting is a configuration set up between the public and private clouds to handle peaks in IT demands. It allows organizations to scale their private infrastructure by borrowing resources from the public cloud when needed, ensuring uninterrupted service.
- **Common Cloud Bursting Situations:**

1. **Software Development:** Temporary need for additional resources during software testing or CI/CD tasks.
2. **Marketing Campaigns:** Handling temporary spikes in web traffic during product launches or promotional events.
3. **Big Data Processing:** Executing one-time queries or generating models that exceed the capacity of the private cloud.
4. **Seasonal Businesses:** Additional computational resources needed during peak times, such as holiday shopping seasons or end-of-business-quarter financial processing.

OpenNebula



Different interfaces provided by OpenNebula

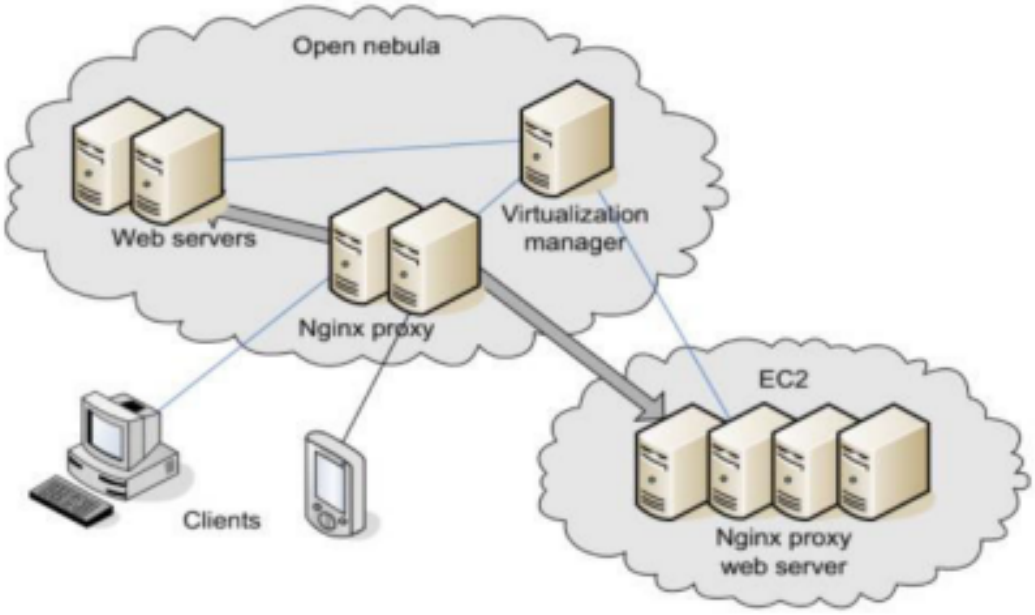
- **Overview:**
  - OpenNebula is an open-source cloud middleware solution designed to manage heterogeneous distributed data center infrastructures.
  - It provides a simple yet feature-rich, production-ready, customizable solution for building and managing enterprise clouds.
  - OpenNebula is easy to install, update, and operate for administrators, and simple to use for end-users.

- **Key Features:**
  - **Heterogeneous Infrastructure Management:** Manages distributed data center infrastructures consisting of various hardware and software configurations.
  - **Multi-Tenancy:** Supports multiple users or tenants, allowing them to share the same cloud infrastructure while maintaining isolation.
  - **Automated Provisioning:** Offers automated provisioning of resources, allowing users to deploy and manage virtual machines and other resources efficiently.
  - **Elasticity:** Provides elasticity by automatically scaling resources up or down based on demand.
  - **Virtual Network Management:** Includes a built-in virtual network manager that maps virtual networks to physical networks.
- **Integration:**
  - OpenNebula combines existing virtualization technologies with advanced features, making it suitable for enterprises.
  - It has been integrated into distributions such as Ubuntu and Red Hat Enterprise Linux.

OpenNebula with Reverse Proxy for Cloud Bursting

- **Architecture:**
  - OpenNebula can be used in conjunction with a reverse proxy to create a cloud bursting hybrid cloud architecture.
  - Load balancing and virtualization support are provided by OpenNebula.
  - The OpenNebula VM controls server allocation in both the EC2 cloud and the OpenNebula cloud.
  - The Nginx proxy distributes the load over the web servers in both the EC2 and OpenNebula clouds.
  - Additionally, the EC2 cloud has its own Nginx load balancer.

By integrating OpenNebula with a reverse proxy, organizations can create a scalable and flexible cloud infrastructure capable of handling variable workloads and ensuring high availability and performance.



Multitenancy

- Multitenancy refers to multiple customers of a cloud vendor using the same computing resources.
- It is an architecture where a single instance of a software application serves multiple customers, often referred to as shared systems.
- A "tenant" represents a group of users who share common access with specific privileges to the software instance.

Multitenancy - Benefits

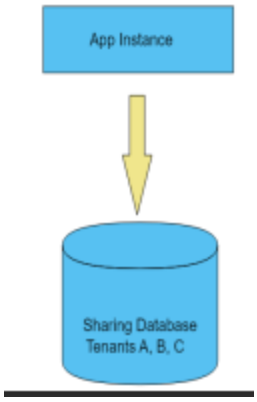
- Multitenancy significantly improves cloud computing by:
    - **Better use of resources:** Sharing machines among multiple tenants maximizes the use of available resources. One machine reserved for one

tenant is not efficient as the tenant is unlikely to use all of the machine's computing power.

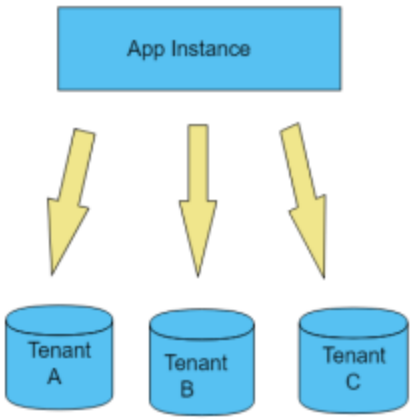
- **Lower costs:** With multiple customers sharing resources, a cloud vendor can offer their services to many customers at a much lower cost compared to each customer requiring their own dedicated infrastructure.

Multi-Tenancy - Architecture Types

1. **Multi-tenancy with a single multi-tenant database:**
   - **Description:** This is the simplest form of multitenancy, using a single application instance and a single database instance to host the tenants and store/retrieve the data.

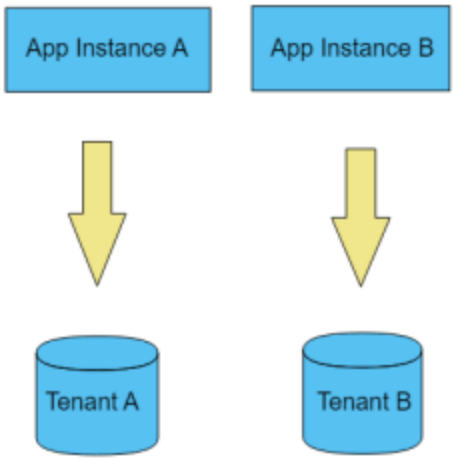


   - **Characteristics:**
     - Highly scalable
     - Low-cost due to shared resources
     - High operational complexity, especially during application design and setup
2. **Multi-tenancy with one database per tenant:**
   - **Description:** This architecture uses a single application instance and an individual database for each tenant.



- **Characteristics:**
  - Scalability may be lower compared to multi-tenancy with a single multi-tenant database.
  - Costs may be higher due to the overhead of each database.
  - Operational complexity is low due to the usage of individual databases.

3. **Standalone single-tenant app with single-tenant database:**
   - **Description:** Each tenant has its own application instance as well as database instance.



- **Characteristics:**
  - Highest level of data isolation

- High cost due to standalone applications and databases

Multi-Tenancy - Requirements

1. **Fine grain resource sharing:**
   - The finer the granularity of sharing, the greater the scalability of the system.
   - Access control needs to be specified for each row of the table to ensure security and isolation between tenants.
2. **Security and isolation between customers:**
   - Security measures must be implemented to ensure that each tenant's data is isolated and protected from unauthorized access by other tenants.
3. **Customization of tables:**
   - The ability to customize tables to accommodate the specific needs of each tenant is essential for providing a tailored experience and optimizing resource usage.

Multi-Tenancy - Example Use Case

- **Shared Machine Method:**
  - Each customer is given their own database process and tables on a shared machine.
- **Shared Process Method:**
  - Each customer has their own database tables, but there is only one database process that executes instructions on behalf of all customers.
- **Shared Table Method:**
  - Customers share the database process, and data is stored in shared tables with each row prefixed with the customer ID to indicate ownership.

Comparison of Methods:

- **Shared Machine Approach:** PostgreSQL used 55MB of main memory and 4MB of disk memory for storing data for one customer.

- **Shared Process Approach:** For 10,000 customers, PostgreSQL used only 80MB of main memory and 4,488MB of disk memory.
- **Observation:** The scalability of the system is much better under the shared process method, demonstrating improved resource utilization and efficiency.

Multi-Tenancy - Levels

Implementing the highest degree of resource sharing for all resources may be expensive in development effort and system complexity. Here are the four levels of multi-tenancy:

1. **Ad Hoc/Custom Instances:**
    - Each customer has their own custom version of the software.
    - Management is difficult as each customer requires their own management support.
2. **Configurable Instances:**
    - All customers share the same version of the program, but customization is possible through configuration and other options.
    - Significant manageability savings over the ad hoc/custom instances level, as there is only one copy of the software to maintain.
3. **Configurable, Multi-Tenant Efficient Instances:**
    - Cloud systems at this level share the same version of the program and have only one running instance shared among all customers.
    - Increased efficiency due to the presence of only one running instance of the program.
4. **Scalable, Configurable, Multi-Tenant Efficient Instances:**
    - In addition to the attributes of the previous level, the software is hosted on a cluster of computers, allowing the system's capacity to scale almost limitlessly.
    - Performance bottlenecks and capacity limitations that may have been present in earlier levels are eliminated.

Multi-Tenancy - Tenants and Users

- The customers of a SaaS or PaaS service are referred to as tenants, regardless of whether they are businesses or individual users.
- The term "user" is used for the actual users of the service.

Multi-Tenancy - Authentication

The key challenge in multi-tenancy is the secure sharing of resources, which can be enforced through authentication. Two approaches can be used:

- **Centralized Authentication System:**
  - All authentication is performed using a centralized user database.
  - The cloud administrator grants the tenant's administrator rights to manage user accounts for that tenant.
  - When the user signs in, they are authenticated against the centralized database.
- **Decentralized Authentication System:**
  - Each tenant maintains their own user database.
  - The tenant deploys a federation service that interfaces between the tenant's authentication framework and the cloud system's authentication service.

Multi-Tenancy - Implementing Resource Sharing

The key technology to ensure secure resource sharing in a multi-tenant service is access control. Two forms of access control can be provided in a cloud service provider:

- **Roles:**
  - Consist of a set of permissions included in the role.
  - For example, a storage administrator role may include the permissions to define storage devices.
  - The ability to specify roles for users is itself a permission that only certain roles can possess.

- **Business Rules:**
    - Policies that provide finer-grained access control than roles provide, as they may depend upon the parameters of the operation.
    - For example, in a banking application, it may be possible to specify a limit on the amount of money a particular role can transfer or specify that transfers can occur only during business hours.
    - Business rules can be implemented using policy engines such as Drools Expert and Drools Guvnor.

Access Control Models:

There are two types of access control models:

1. **Access Control Lists (ACL):**
    - Each object is attached with a list of permissions for each role.
2. **Capability-Based Access Control:**
    - Works like a house key. If a user holds a reference or capability to an object, they have access to the object.
    - The key is the link to the object; the virtue of the user having this key grants them access to the object.

Multi-Tenancy - Resource Sharing

Sharing Storage Resources:
In a multi-tenant system, many tenants share the same storage system. Cloud applications use two kinds of storage systems:

1. **File Systems and Databases:**
    - File systems are shared to specified users via ACLs and other mechanisms.
    - Data in a single database can be shared using two methods:
        - Table sharing
        - Dedicated tables per tenant
- **Dedicated Table Method:**

- Each tenant stores their data in separate sets of tables, restricting access to users.

- **Shared Table Approach:**
  - Data for all tenants is stored in the same table in different rows.
  - Space-efficient but requires multiple SELECT statements.
  - An auxiliary table, called a metadata table, stores information about the tenants.

Sharing Compute Resources:

1. In the dedicated table method, each tenant has their own set of files, ensuring that one tenant cannot read the tables of another tenant.
2. In the shared table case, the cloud system relies upon the application to ensure the security of the data.

Customization:
Cloud infrastructure should provide customization of stored data. Three methods are:

1. **Preallocated Columns:**
   - Space is reserved in the tables for custom columns, which can be used by tenants for defining new columns.
   - However, there could be a lot of wasted space with this method.
2. **Name-Value Pair:**
   - An extra column acts as a pointer to a table of name-value pairs.
   - Space-efficient but requires data reconstruction before it is used by joins.
3. **XML Method:**
   - The final column is an XML document where records of any arbitrary structure can be stored.

Failure detection – Checkpointing & Application recovery

Failure detection is valuable to distributed systems as it adds to their reliability and increases their usefulness.

- Distributed systems should be able to detect and cater to failures efficiently and accurately when they occur.
- Consensus cannot be solved in a totally asynchronous model, but by augmenting the asynchronous system model with failure detectors, it would be possible to bypass this impossibility.
- Failure detection should be a component of the operating system.
- All failure detection algorithms/schemes use time as a means to identify failure. Different protocols/ways of using this tool vary on how the timeout issue should be addressed, when/where messages should be sent, and how often, synchronous or asynchronous.
- For small distributed networks, such as LANs, coordination and failure detection are simple and do not require much complexity.

An independent failure management system should have the following three functional modules:

1. A library that implements simple failure management functionality and provides the API to the complete service.
2. A service implementing per-node failure management, combining fault management with other local nodes to exploit the locality of communication and failure patterns.
3. An inquiry service closely coupled with the operating system which, upon request, provides information about the state of local participating processes.

Support for High Availability:

Two types of support can be built into the cloud infrastructure for high availability:

1. **Failure Detection:**
    - The cloud infrastructure detects failed application instances and avoids routing requests to such instances.
2. **Application Recovery:**
    - Failed instances of applications are restarted.

Failure Detection - Introduction

Cloud providers, such as Amazon Web Services' Elastic Beanstalk, detect when an application instance fails and avoid sending new requests to the failed instance. To detect failures, monitoring for failures is necessary.

1. **Failure Monitoring:**

   There are two techniques for failure monitoring:

   - **Heartbeats:** An application instance periodically sends a signal (called a heartbeat) to a monitoring service in the cloud. If the monitoring service does not receive a specified number of consecutive heartbeats, it may declare the application instance as failed.
   - **Probes:** Probes periodically send a lightweight service request to the application instance. If the instance does not respond to a specified number of probes, it may be considered failed.

The two most famous strategies for failure detection are the push and pull models:

1. The Push Model:

Assuming we have two processes, p and q, with p being the monitor process:

- Process q sends heartbeat messages every t seconds.
- Process p expects an "I am alive" message from q every t seconds.
- If after a timeout period T, p does not receive any messages from q, it starts suspecting q has failed.

2. The Pull Model:

Assuming the same parameters as the Push model:

- In this model, instead of process q (the one that needs to prove it's alive) sending messages to p every few seconds, it sends "are you alive?" messages to q every t seconds and waits for q to respond with "yes, I am alive."
- If q crashes, p does not receive any more responses from q and starts suspecting q has failed after a timeout period of T.

The Dual Scheme:

The pull model is somewhat inefficient as there are potentially too many messages sent between processes. The push model is a bit more efficient in this manner. Hence, a model is proposed that is a mix of the two.

- During the first message-sending phase, any q-like process that is being monitored by p is assumed to use the push model and hence send "I am alive" or "liveness" messages to p.
- After some time, p assumes that any q-like process that did not send a liveness message is using the pull model.

Trade-off between speed and accuracy:

There is a trade-off between the speed and accuracy of detecting failures. To detect failures rapidly, set a low value for the number of missed heartbeats or probes. However, this could lead to an increase in the number of false failures. An application may not respond due to a momentary overload or some other transient condition. Since falsely declaring an application instance failed can have severe consequences, generally, a high threshold is set for the number of missed heartbeats or probes.

Redirection:

After identifying failed instances, it is necessary to avoid routing new requests to these instances.

- A common mechanism used in HTTP-based protocols is HTTP-redirection.
- The Web server returns a 3xx return together with a new URL to be visited. For example, if a user types “http://www.pustakportal.com/” into their browser, the request may first be sent to a load-balancing service at Pustak Portal, which may return a return code of 302 with a URL “http://pps5.pustakportal.com”.
- pp5.pustakportal.com is the address of a server that is currently known to be up, and the user is directed to a server that is up.

Application Recovery

In addition to directing new requests to a server that is up, it is necessary to recover old requests.

- **Check-point/restart:** The cloud infrastructure periodically saves the state of the application. If the application has failed, the most recent check-point can

be activated, and the application can resume from that state.

- **Distributed checkpoint/restart:** In this method, all processes of distributed application instances are checkpointed, and all instances are restarted from a common checkpoint if any instance fails. However, this method has scalability limitations and also suffers from correctness issues if any inter-process communication data is in transit at the time of failure.

Cloud Security Requirements

Cloud Security

Trust and security become even more demanding for web and cloud services, as leaving user applications completely to the cloud providers has faced strong resistance from most PC and server users. Cloud platforms become worrisome to some users due to the lack of privacy protection, security assurance, and copyright protection.

What is Cloud Security?

Cloud security is a set of control-based safeguards and technology protection designed to protect resources stored online from leakage, theft, or data loss. This protection encompasses cloud infrastructure, applications, and data from threats.

Cloud Security Requirements - Physical and Virtual Security

At a high level, the cloud infrastructure can be partitioned into physical infrastructure and virtual infrastructure. Similarly, security requirements and best practices can be divided into the requirements for physical security and those for virtual security.

Basic Objectives of Cloud Security:

- Confidentiality
- Integrity
- Availability of the cloud system

Additional Objectives:

- Cost-effectiveness
- Reliability & performance

Physical Security

Physical security implies that the data center where the cloud is hosted should be secure against physical threats. This includes not only attempts at penetration by intruders but also protection against natural hazards and disasters such as floods, and human errors such as switching off the air conditioning. It's important to note that security is only as strong as its weakest link.

Cloud Security Requirements

To ensure physical security, a multi-layered system is required, which includes:

1. A central monitoring and control center with dedicated staff.
2. Monitoring for each possible physical threat, such as intrusion or natural hazards such as floods.
3. Training of the staff in response to threat situations.
4. Manual or automated backup systems to help contain threats (e.g., pumps to help contain the damage from floods).
5. Secure access to the facility. This requires that various threats to the data center be identified, and appropriate procedures derived for handling these threats.

Virtual Security

The following best practices have been found to be very useful in ensuring cloud security:

1. **Cloud Time Service**:
    - If all systems in the data center are synchronized to the same clock, it helps ensure correct operation of the systems and facilitates later analysis of system logs.
    - Network Time Protocol (NTP) is commonly used for this purpose, which synchronizes the clock on a computer to a reference source on the Internet. Protocol messages can be encrypted to protect against false reference sources.
2. **Identity Management**:

- Identity management is a foundation for achieving confidentiality, integrity, and availability.
- Requirements for identity management include scalability, federated identity management system for heterogenous cloud systems, compliance with applicable legal and policy requirements, and maintenance of historical records for possible future investigation.

3. **Access Management**:
    - The core function of access management is to allow access to cloud facilities only to authorized users.
    - Additional requirements include not allowing unrestricted access to cloud management personnel, implementation of multi-factor authentication, disallowing shared accounts, and implementing IP address white-listing for remote administrative actions.
4. **Break-Glass Procedures**:
    - The access management system should allow alarmed break-glass procedures, which bypass normal security controls in emergency situations.
    - These procedures should be executable only in emergencies under controlled situations and should trigger an alarm.
5. **Key Management**:
    - Encryption is a key technology to ensure isolation of access in a cloud with shared storage.
    - Cloud infrastructure needs to provide secure facilities for the generation, assignment, revocation, and archiving of keys.
    - Procedures for recovering from compromised keys should be generated.
6. **Auditing**:
    - The audit should capture all security-related events, together with data needed to analyze the event.
    - The audit log should be centrally maintained, secure, and possible to sanitize or produce a stripped-down version for sharing with cloud

customers if their assistance is needed to analyze the logs.

7. **Security Monitoring**:
    - This includes an infrastructure to generate alerts when a critical security event has occurred.
    - Intrusion detection systems may be installed both on the network and the host nodes.
    - Cloud users may need to implement their own intrusion and anomaly detection systems.
8. **Security Testing**:
    - All software should be tested for security before deployment in an isolated test bed.
    - Patches to software should also be tested before being released into production.
    - Ongoing security testing should be carried out to identify vulnerabilities in the cloud system.
    - Depending upon the risk assessment, some of these tests may be carried out by third parties, and there should be a remediation process to fix identified vulnerabilities.

Risk Management & Security Design Patterns

Why use a risk approach for cloud selection?

Many organizations are embracing cloud computing due to its popularity. However, there are several reasons why a risk approach should be used for cloud selection:

1. **Data security risks**: Entrusting sensitive data to an external third party raises concerns about data security.
2. **Preparation for cloud failure**: Cloud outages, such as those experienced by Microsoft and Amazon, highlight the importance of being prepared for cloud

failures.

- In March 2009, Microsoft Windows Azure was down for 22 hours.
- In April 2011, a large-scale outage affected Amazon Web Services' Elastic Compute Cloud (EC2).
- These outages prevent users from accessing applications or data stored in the cloud, resulting in potentially high financial costs, especially for mission-critical systems.

3. **Recent examples of cloud outages**:
    - On August 25, 2013, Amazon Web Services (AWS) experienced an outage for 59 minutes due to issues with its U.S.-EAST data center.
    - This outage led to a degraded experience, with a small number of EC2 instances becoming unreachable due to packet loss.
    - The outage affected Amazon.com, resulting in rejected customer access to its site in the U.S. and Canada. Other Amazon-owned websites, including Audible.com, were also affected.

What is Risk Management?

Risk management is the process of evaluating risks, deciding how they are to be controlled, and monitoring their operation. When managing risk, several factors need to be considered:

- Suitable approaches in different domains (e.g., finance and healthcare) may vary.
- There should be a careful trade-off between the impact of the risks involved and the cost of security measures, measured in terms of both the impact on system usage and the actual cost of security.

Risk Management Concepts

- **Security controls**: Safeguards (processes or system functions) to prevent, detect, or respond to security risks. NIST divides security controls into three broad categories: Technical, Operational, and Management.
- **FIPS 200 security requirement**: Defines system security requirements as low-impact, moderate-impact, or high-impact, depending on the impact of a

security breach.

- Low-impact systems: Experience limited degradation in capability but can still perform primary functions.
- Moderate-impact systems: Experience significant degradation in functions but can still perform primary functions.
- High-impact systems: Are incapable of performing some primary functions after a security breach.

Risk Management Process

1. **Information Resource Categorization**:
   - Evaluate each information resource in the organization based on:
     a.
     Criticality: Impact to the business in case of a security failure.
     b.
     Sensitivity: Confidentiality level of the resource.
   - Determine the level of security needed for each resource based on this evaluation.
2. **Select Security Controls**:
   - Choose security controls appropriate to the criticality and sensitivity of the information resources.
   - Example: Whitelisting IP addresses as a security control may be suitable for restricting access to certain areas. However, it might not be appropriate for user networks, as it would require users to register new IP addresses each time they access the network from a new location.
3. **Risk Assessment**:
   - Evaluate whether the selected security controls provide adequate protection against anticipated threats.
   - Augment the controls if additional protection is required.
4. **Implement Security Controls**:

- Implement the selected security controls, which may include administrative, technical, or physical measures.

5. **Operational Monitoring**:
   - Continuously monitor the effectiveness of the implemented security controls.
6. **Periodic Review**:
   - Regularly review the security controls to ensure they remain effective.
   - Reasons for review:
     a. New threats may emerge.
     b. Operational changes (e.g., new software) may require adjustments to the security design.

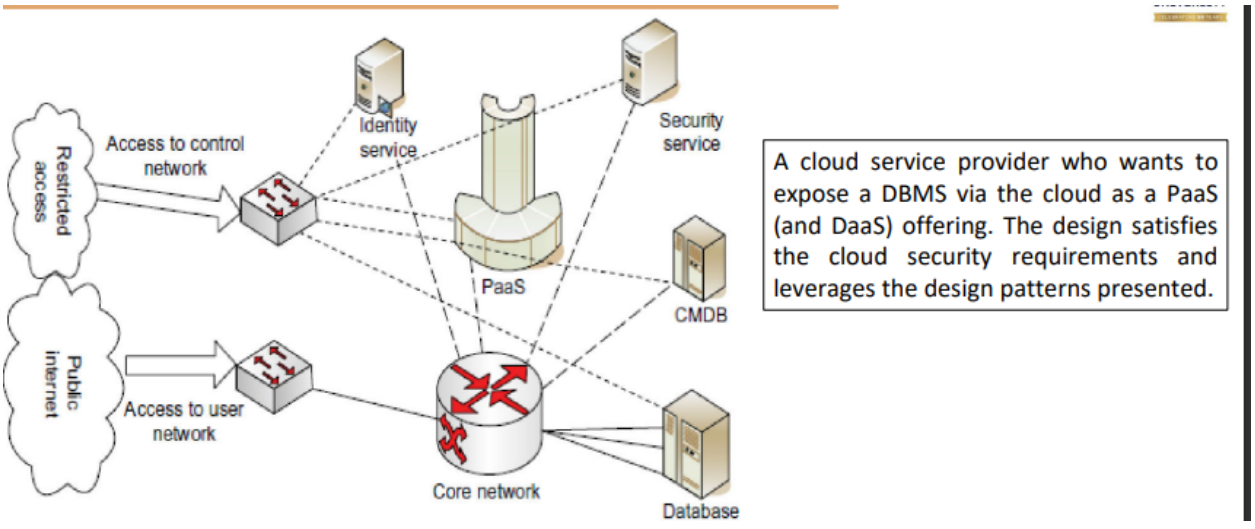
Security Design Patterns

1. **Defense in Depth**:
   - Layered defenses to require attackers to overcome multiple obstacles.
   - Example: Allowing remote administrative access only through a VPN, restricted to whitelisted IP addresses, and requiring a one-time password for additional security.
2. **Honeypots**:
   - Decoy systems designed to attract attackers, enabling security personnel to observe and control attacks.
   - Example: Deploying a honeypot virtual machine in the cloud to monitor and trap suspicious attempts to access it.
3. **Sandboxes**:
   - Execution of software in a restricted environment within the operating system, limiting access to system resources.
   - Example: Executing software in a sandbox to restrict an attacker's access to the operating system and minimize potential damage.
4. **Network Patterns**:

- **VM Isolation**:
  - Techniques to isolate traffic between virtual machines (VMs) on the same physical hardware.
  - Security features include encrypted traffic between VMs and tightened security controls on VMs.
- **Subnet Isolation**:
  - Physically separating traffic for administrative, customer, and storage networks to prevent misconfigurations.
  - Routing between networks is handled by firewalls.

5. **Common Management Database (CMDB)**:
   - A database containing information about IT system components, configurations, and status.
   - Simplifies infrastructure management, including security, by providing a single consistent view of all components.

Example - Security Design for a PAAS System:



- **External Network Access**:
  - Control network access and public network access are separate and lead to distinct physical networks.

    - Control network access limited to whitelisted IP addresses, and multi-factor authentication is mandatory for secure access to administrative functions.
- **Internal Network Access**:
    - Management network is physically separated from the public network to reduce the risk of unauthorized access.
    - The DBMS is connected to the public network via aggregated links to ensure increased bandwidth and availability.
- **Server Security**:
    - Access to cloud services managed through an identity server and access control.
    - Additional security measures include disallowing access to unneeded ports, implementing intrusion detection systems, and monitoring ODBC connections to the database.
- **Security Server**:
    - Performs security services such as auditing, monitoring, hosting a security operations center, and security scanning of the cloud infrastructure.

Security Architecture ,Legal & Regulatory Issues

Security Architecture Standards

1. **SSE-CMM**:
    - System Security Engineering Capability Maturity Model, adapted from the Capability Maturity Model (CMM) for software engineering by Carnegie Mellon University.
    - Defines five capability levels for organizations to assess and improve their security processes.
2. **ISO/IEC 27001-27006**:

- Set of standards providing an Information Security Management System (ISMS).
- Specifies requirements organizations must satisfy, such as systematically evaluating information security risks.

3. **European Network and Information Security Agency (ENISA)**:
   - Offers the Cloud Computing Information Assurance Framework to assess cloud service risks, compare providers, and reduce assurance burdens.
   - Based on ISO/IEC 27001-27006 and includes an assessment of cloud computing benefits and risks.
4. **ITIL Security Management**:
   - Based on Information Technology Infrastructure Library (ITIL) standards for IT service management.
   - Security management section based on ISO/IEC 27002, providing an integrated security solution with a smaller learning curve for ITIL users.
5. **COBIT**:
   - Control Objectives for Information and Related Technology developed by ISACA (International organization for IT governance standards).
   - Defines processes and best practices linking business goals to IT goals, along with metrics and a maturity model.
6. **NIST**:
   - US National Institute of Standards and Technology provides whitepapers and resources through its Security Management & Assurance working group.
   - Offers recommendations primarily targeted at U.S. federal agencies but applicable to other organizations as well.

Legal and Regulatory Issues in Cloud Computing

1. **Third-party/Contractual Issues**:
   - **Due diligence**:

    - Define the scope of cloud services required and specify regulations and compliance standards.
    - Evaluate cloud service providers to ensure they can meet applicable laws and regulations.
    - Assess risks related to the stability and reliability of the cloud service provider.
- **Contract negotiation**:
    - Negotiate contracts with cloud service providers to ensure security measures are included.
    - Standardized agreements may be acceptable for low-risk scenarios.
    - Cloud service providers may use external accreditations like Statement on Auditing Standards (SAS) 70.
- **Implementation**:
    - Ensure that the security safeguards laid out in the contract are followed.
    - Continuous evaluation of the system to check for changes in circumstances.
- **Termination**:
    - Identify alternative service providers for a smooth transition in case of contract termination.
    - Ensure timely and secure transfer of services to the new provider.
    - Sensitive data should be deleted from the original service provider's system.

2. **Data Handling**:
    - **Data Privacy**:
        - Protect the privacy of collected data and use it only for the intended purpose.
        - Ensure compliance with privacy laws and regulations.

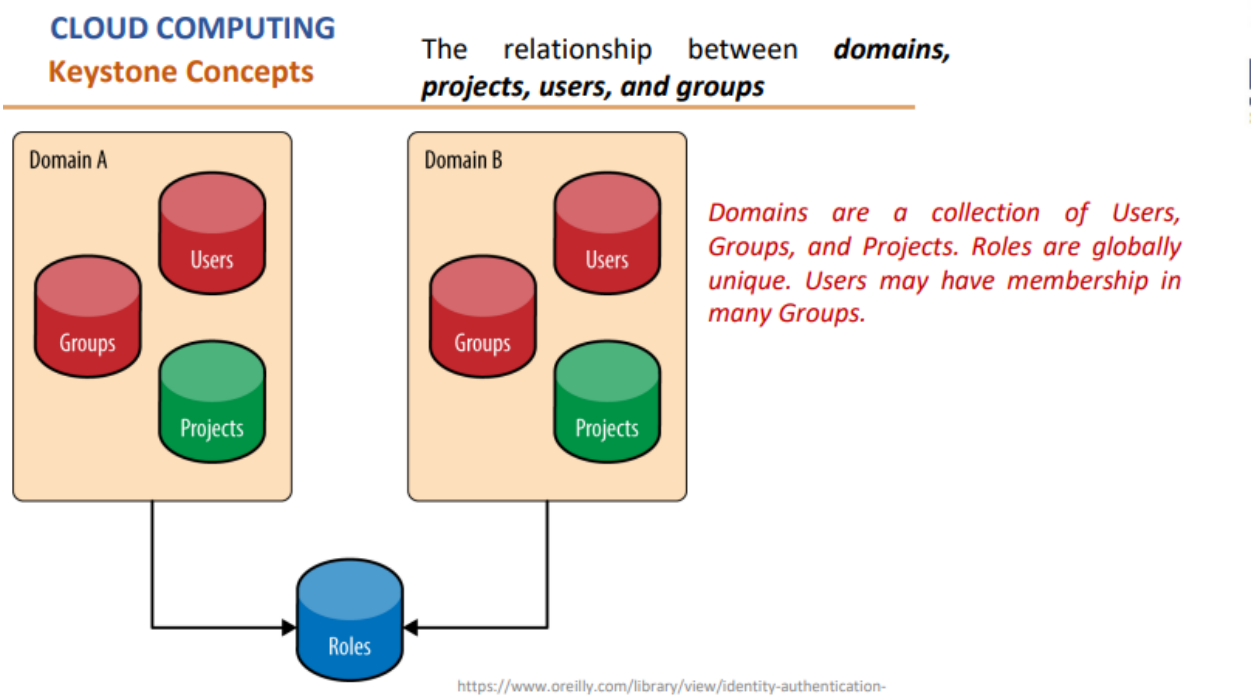
    - Subcontractors, including cloud service providers, must adhere to these regulations.
- **Data Location**:
    - Understand the laws related to data handling in different countries.
    - Specify the location of data centers and backups to comply with legal requirements.
    - Amazon's storage services allow the specification of a region, governing the location of data storage.
- **Secondary Use of Data**:
    - Prevent unauthorized access to data and ensure that cloud service providers do not use the data for data mining or other secondary purposes.
    - Review service agreements carefully before accepting.
- **Business Continuity Planning and Disaster Recovery**:
    - Expand Business Continuity Planning (BCP) and Disaster Recovery (DR) plans to include cloud service provider-related catastrophes.
    - Utilize features provided by the cloud service provider for disaster recovery, such as multiple data locations.

3. **Security Breaches**:
    - Establish notification procedures for security breaches.
    - Ensure that service agreements specify actions to be taken during a breach.
4. **Litigation Related Issues**:
    - Ensure that cloud providers can respond promptly to data requests during litigation.
    - Cloud providers should notify businesses promptly if asked to provide data during litigation, allowing the business to contest the request if necessary.

Authentication in cloud – Keystone

Keystone Concepts

Keystone is an OpenStack service that provides API client authentication, service discovery, and distributed multi-tenant authorization by implementing OpenStack's Identity API



1. **Project**:
    - In Keystone, a Project is an abstraction used by other OpenStack services to group and isolate resources like servers, images, etc.
    - Keystone serves as the registry of Projects and manages access to those Projects.
    - Projects themselves don't own Users; instead, Users or User Groups are given access to a Project using Role Assignments.
    - Role Assignments denote that the User or User Group has access to resources in the Project, and the assigned Role determines the type of access and capabilities.
2. **Domain**:

- Keystone introduced the Domain abstraction to support multiple user organizations simultaneously.
- A Domain provides the ability to isolate the visibility of a set of Projects and Users (and User Groups) to a specific organization.
- Formally, a Domain is defined as a collection of users, groups, and projects.
- Domains enable the division of cloud resources into silos for specific organizations, serving as a logical division within an enterprise or between separate enterprises.

3. **Users and User Groups (Actors)**:
    - Users and User Groups are entities given access to resources isolated in Domains and Projects.
    - Groups are collections of Users.
    - Users are individuals who will use the cloud.
    - Users and Groups are referred to as "Actors" since, when assigning a role, these are the entities to which the role is assigned.
4. **Roles**:
    - Roles are used in Keystone for Authorization.
    - An actor may have multiple roles on a target. For example, the role of admin is assigned to the user "bob" and it is assigned on the project "development".
5. **Assignment**:
    - A role assignment consists of an actor, a target, and a role.
    - Role assignments are granted and revoked and may be inherited between groups, users, domains, and projects.
6. **Targets**:
    - Projects and Domains are entities where roles are assigned.
    - Users or User Groups are given access to Projects or Domains by assigning a specific Role value for that User or User Group for a specific

Project or Domain.

- Both Projects and Domains are collectively referred to as Targets.

7. **Tokens**:
    - Users need to prove their identity and authorization to call any OpenStack API. They do this by passing an OpenStack token into the API call.
    - Keystone generates these tokens upon successful user authentication.
    - Tokens carry authorization information and have both an ID and a payload.
    - The payload contains information about the token's creation time, expiration time, the authenticated user, and the project the token is valid on.


```
"token": {
        "issued_at": "201406-10T20:55:16.806027Z",
        "expires_at": "2014-06-10T2:55:16.806001Z",
        "roles": [{
        "id": "c703057be878458588961ce9a0ce686b",
        "name": "admin"}
        ],
```


8. **Catalog**:
    - The service catalog is essential for an OpenStack cloud, containing the URLs and endpoints of different Cloud services.
    - It enables users and applications to know where to route requests for creating VMs or storing objects.
    - The service catalog is broken into a list of endpoints, and each endpoint contains an admin URL, internal URL, and public URL.

These concepts are crucial for understanding Keystone's role in OpenStack, providing API client authentication, service discovery, and distributed multi-tenant authorization.

Identity Service in Keystone

The Identity Service in Keystone provides Actors. Identities in the Cloud can come from various sources, including SQL, LDAP, and Federated Identity Providers.

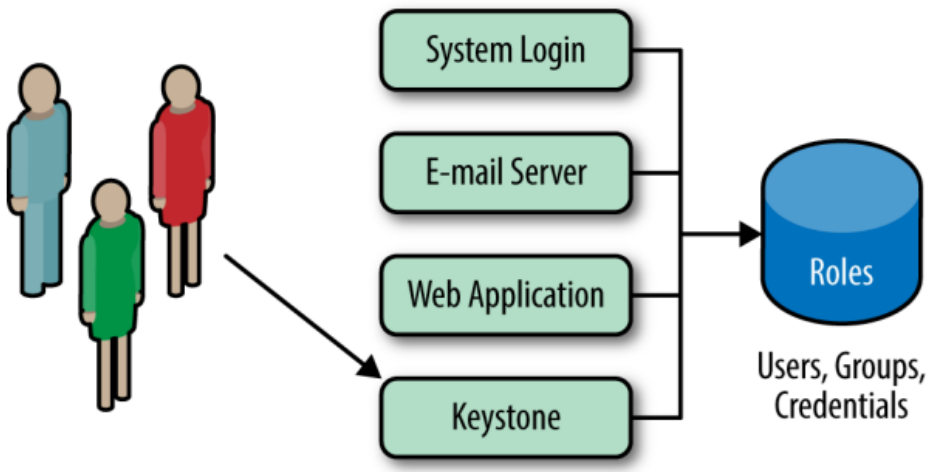
1. **SQL**:
    - Keystone includes the option to store users and groups (Actors) in SQL databases like MySQL, PostgreSQL, and DB2.
    - Information such as name, password, and description is stored in the database.
    - Configuration settings for the database must be specified in Keystone's configuration file.

    Pros:
    - Easy to set up.
    - User and group management through OpenStack APIs.

    Cons:
    - Keystone should not be an Identity Provider.

- Weak password support:
    - No password rotation or recovery.
- Most enterprises prefer using LDAP servers.
- Identity silo: Another set of credentials for users to remember.

2. **LDAP**:
    - Keystone can retrieve and store users and groups in Lightweight Directory Access Protocol (LDAP).
    - LDAP is accessed like any other application using LDAP for authentication.
    - Configuration for connecting to LDAP is specified in Keystone's configuration file.
    - Ideally, LDAP should perform only read operations like user and group lookup and authentication.



Keystone should use an internal LDAP just like any other application

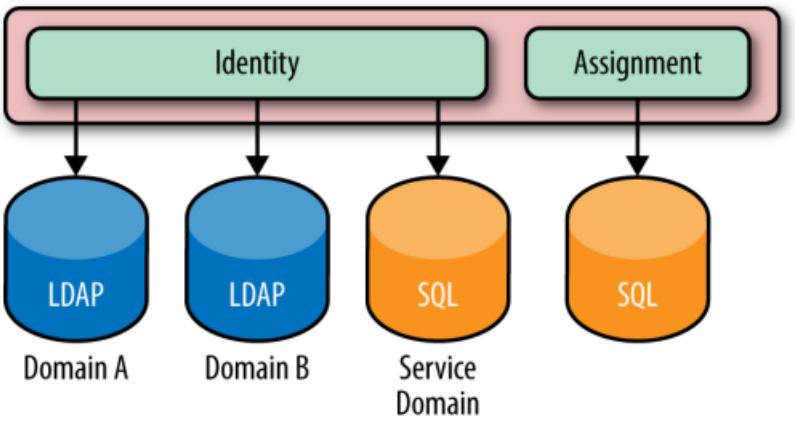
Pros:

- No need to maintain copies of user accounts.
- Keystone does not act as an Identity Provider.

Cons:

- Service accounts still need to be stored somewhere.
- Keystone has visibility of user passwords during authentication, which is not ideal.

Multiple Backends in Keystone Identity



The Identity service may have multiple backends per domain. LDAPs for both Domains A and B, and an SQL-based backend for the service accounts is usual. The Assignment service is also shown to remind readers that the assignment, resource, and identity service may all be backed by various stores.

https://www.oreilly.com/library/view/identity-authentication-

Keystone supports Multiple Backends, allowing a deployment to have one identity source (backend) per Keystone domain.

Motivation:

- In enterprise settings, LDAP administrators may be different from the OpenStack deployment team. Hence, LDAP is typically restricted to employee information only.
- Multiple Identity backends and domains allow the use of multiple LDAPs. This is beneficial for scenarios like company mergers or when different departments have different LDAPs.
- Multiple backends enable enterprises to use LDAP for user accounts and SQL for service accounts, maintaining a logical split between identity backends and domains.

Implementation:

- The Identity service can have multiple backends per domain, such as LDAPs for different domains and an SQL-based backend for service accounts.
- The Assignment service may also use various stores, as shown in the architecture diagram.

Pros:

- Simultaneous support for multiple LDAPs for user accounts and SQL backends for service accounts.
- Leverage existing identity LDAP without impacting it.

Cons:

- Slightly more complex setup.
- Authentication for user accounts must be domain-scoped.

Identity Providers:

Keystone can consume federated authentication via Apache modules for multiple trusted Identity Providers. These users are not stored in Keystone and are treated as ephemeral. Federated users have their attributes mapped into group-based role assignments.

Identity Providers Pros:

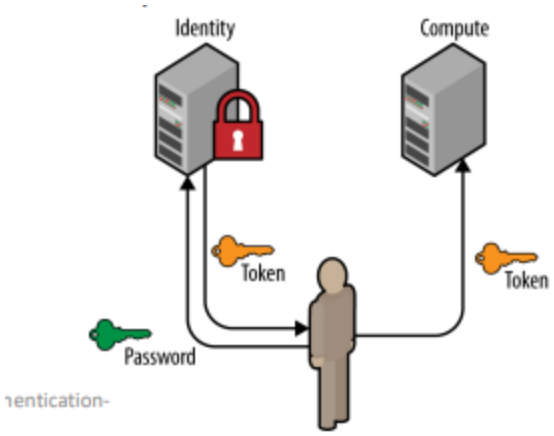
- Leverage existing infrastructure and software for user authentication and information retrieval.
- Further separation between Keystone and handling identity information.
- Opens possibilities for federation, such as single sign-on and hybrid cloud.
- Keystone doesn't have access to user passwords.
- Identity provider handles authentication completely, making the authentication method irrelevant to Keystone.

Identity Providers Cons:

- Most complex setup of the identity sources.

Authentication-Password:

1. **User Identification and Authentication:**
    - The payload of the authentication request must contain enough information to:
        - Identify where the user exists.
        - Authenticate the user.
        - Optionally, retrieve a service catalog based on the user's permissions.



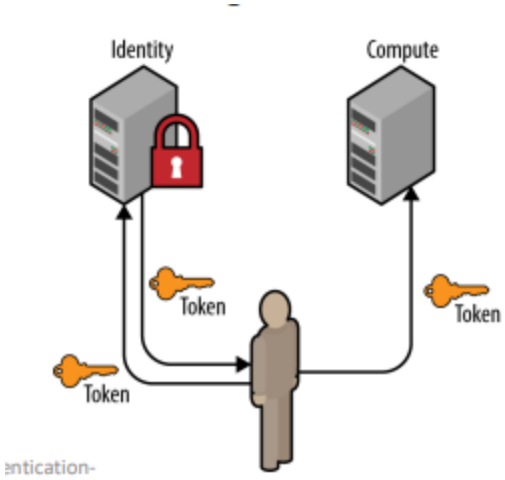
1. **User Section:**
    - It identifies the incoming user.
    - Should include domain information (either the domain name or ID).
    - If using a globally unique ID for the user, it's sufficient for identification.
    - Proper scoping is necessary, especially in a multi-domain deployment where there may be multiple users with the same name.
2. **Scope Section (Optional but often used):**
    - Used to indicate which project the user wishes to work with.
    - Without a scope, a user will not be able to retrieve a service catalog.
    - Must have enough information about the project to find it, so the owning domain must be specified.

```
{
  "auth": {
    "identity": {
      "methods": [
        "password"
      ],
      "password": {
        "user": {
          "domain": {
            "name": "example.com"
          },
          "name": "Joe",
          "password": "secretsecret"
        }
      }
    },
    "scope": {
      "project": {
        "domain": {
          "name": "example.com"
        },
        "name": "project-x"
      }
    }
  }
}
```


An example of an authentication payload request that contains a scope.

https://www.oreilly.com/library/view/identity-authentication-and/9781491941249/ch01.html

Authentication-Token:

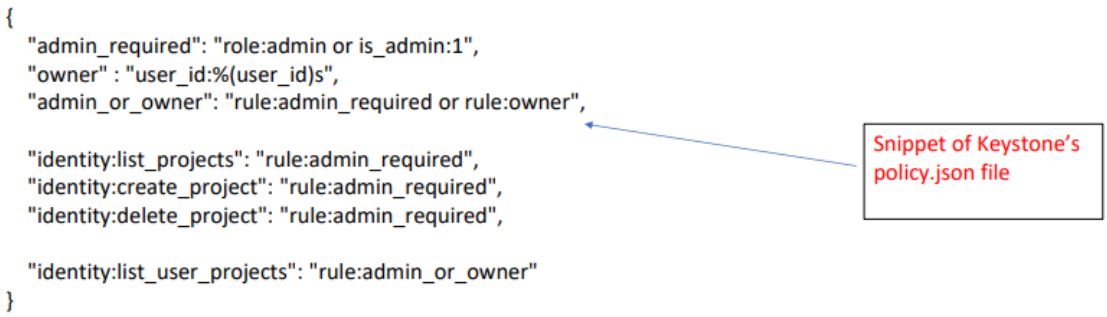


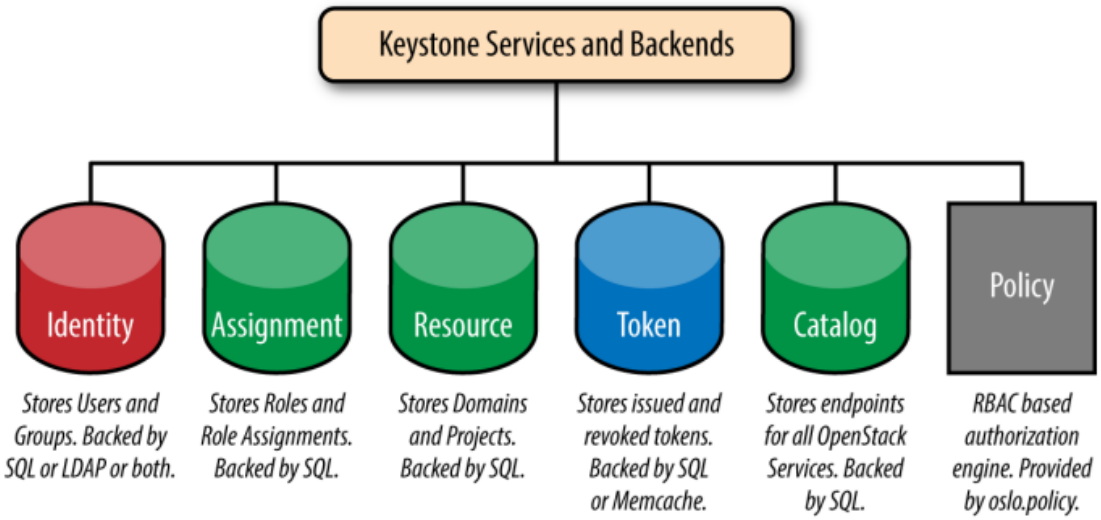
1. **Token-Based Authentication:**
    - Similar to using a password, a user can request a new token by providing a current token.
    - The payload of this request is significantly shorter compared to using a password.
2. **Reasons for Using Tokens:**
    - Tokens are used to retrieve another token for various reasons, such as:
        - Refreshing a token that is about to expire.

- Changing an unscoped token to a scoped token.

Access Management and Authorization:

1. **Role-Based Access Control (RBAC):**
   - Keystone manages access and authorizes which APIs a user can use through Role-Based Access Control (RBAC) policies.
   - These policies are enforced on each public API endpoint.
2. **Policy Definition:**
   - RBAC policies are stored in a file on disk, commonly named `policy.json`.
   - Targets and rules are defined in this file.
   - Targets refer to the left-hand key, and rules refer to the right-hand value.
   - At the top of the file, targets are established for evaluation of other targets.
   - This is where the definition of roles like admin, owner, or others are defined.





Summary of Keystone

Keystone: Managing Identity and Access in OpenStack

1. Identity Management:

- Keystone is an OpenStack service responsible for managing identity, authentication, and access control within OpenStack.
- It provides authentication, service discovery, and multi-tenant authorization by implementing OpenStack's Identity API.

2. Concepts in Keystone:

a. Projects and Domains:

- **Projects:** In Keystone, a Project is used to group and isolate resources like servers and images. Users or User Groups are given access to Projects through Role Assignments.
- **Domains:** Domains isolate the visibility of Projects and Users to specific organizations. They serve as logical divisions between different portions of an enterprise or different enterprises.

b. Users and User Groups:

- **Users:** Individuals who use the cloud. Users are given access to resources within Domains and Projects.
- **User Groups:** Collections of Users.

c. Roles and Role Assignments:

- **Roles:** Used to convey authorization in Keystone. An actor may have numerous roles on a target, specifying the type of access and capabilities.
- **Role Assignments:** The combination of an actor, a target, and a role. Role assignments are granted and revoked and may be inherited between groups, users, domains, and projects.

d. Tokens:

- Tokens are used for authentication in OpenStack. They are provided by Keystone upon successful authentication.
- Tokens contain authorization information, including the user's permissions on a scope (project).

e. Multiple Backends:

- Keystone supports multiple identity backends per domain. It allows a deployment to have one identity source (backend) per Keystone domain.
- Multiple backends enable enterprises to use LDAP for user accounts and SQL for service accounts, maintaining a logical split between identity backends and domains.

f. Identity Providers:

- Keystone can consume federated authentication via trusted Identity Providers.
- Federated users are not stored in Keystone and are treated as ephemeral. Keystone maps their attributes into group-based role assignments.

3. Authentication:

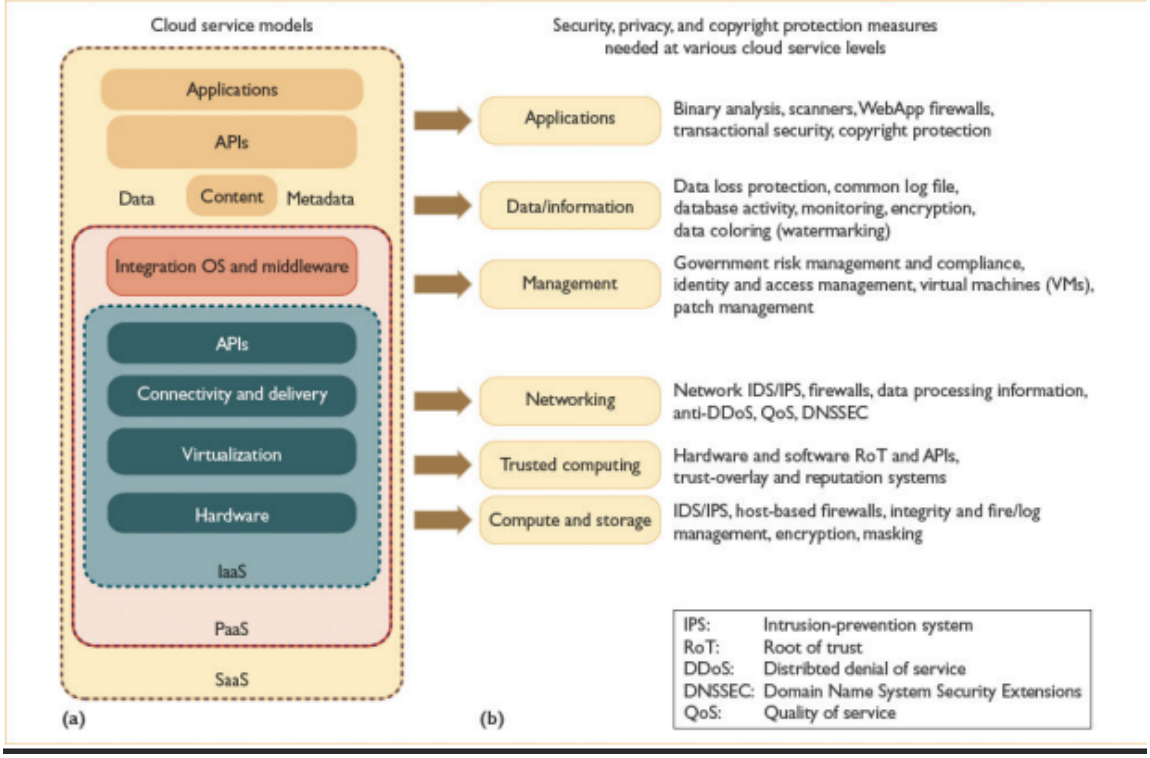
- Users can authenticate with Keystone using passwords or tokens.
- **Password Authentication:** Users provide their username and password in a POST request to Keystone's authentication endpoint.
- **Token Authentication:** Users can request a new token by providing a current token. Tokens are used for various purposes, including refreshing an expiring token or changing an unscoped token to a scoped token.

4. Access Management and Authorization:

- Keystone enforces Role-Based Access Control (RBAC) policies on each public API endpoint.
- RBAC policies are stored in a file named `policy.json` and define which APIs a user can access based on their role.
- Targets and rules are defined in the `policy.json` file, where roles like admin, owner, or others are specified.

In summary, Keystone plays a crucial role in OpenStack by managing identity, authentication, and access control. It provides a flexible and secure system for authenticating users, managing access to resources, and enforcing access control policies through Role-Based Access Control (RBAC).

Cloud Threats –Dos & EDoS



1. What is Cloud Security?

- Cloud security comprises control-based safeguards and technology protections designed to safeguard resources stored online from leakage, theft,

or data loss. It includes protection for cloud infrastructure, applications, and data from various threats.

2. Cloud Security Issues:

- Trust and security are significant concerns for web and cloud services due to fears regarding privacy protection, security assurance, and copyright protection.

3. Cloud Security Requirements:

- Cloud security defense strategies aim to protect against abuses, violence, cheating, hacking, viruses, rumors, pornography, spam, privacy, and copyright violations.
- Security demands vary for different cloud service models (IaaS, PaaS, and SaaS) based on Service Level Agreements (SLAs) between providers and users.

4. Basic Cloud Security Enforcements:

- Facility Security:
  - On-site security in data centers, including biometric readers, CCTV, motion detection, and man traps.
- Network Security:
  - Fault-tolerant external firewalls, intrusion detection systems (IDSes), third-party vulnerability assessment.
- Platform Security:
  - SSL and data encryption, strict password policies, system trust certification.

5. Security Challenges in Virtual Machines (VMs):

- Attacks may result from hypervisor malware, guest hopping, hijacking, VM rootkits, and man-in-the-middle attacks for VM migrations.
- Active attacks may manipulate kernel data structures, causing significant damage to cloud servers.

6. Cloud Defense Methods:

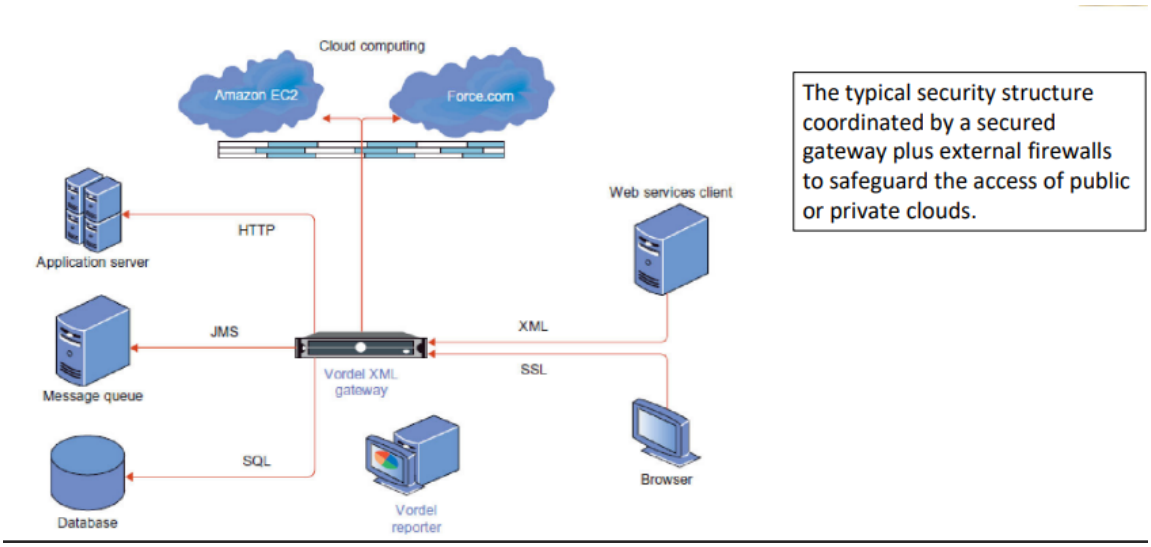
- Program Shepherding:
  - Controls and verifies code execution to prevent malware attacks.
- Virtualization:
  - Provides better security isolation for VMs, protecting them from DoS attacks and containing security attacks within one VM.
  - Trust negotiation at the SLA level, Public Key Infrastructure (PKI) services augmented with data-center reputation systems, containment of worm and DDoS attacks.

7. Defense with Virtualization:

- VMs are decoupled from physical hardware, allowing easy saving, cloning, encryption, moving, or restoration.
- Enables High Availability and faster disaster recovery.
- Live migration of VMs can be used to build distributed intrusion detection systems (DIDSes) for multiple IDS VMs deployed at various resource sites.

8. Privacy and Copyright Protection:

- Shared files and data sets in cloud computing environments raise concerns about privacy, security, and copyright protection.
- Desired security features include dynamic web services with secure web technologies, trust between users and providers through SLAs and reputation systems, effective user identity management and data-access management, single sign-on and single sign-off, auditing, copyright compliance, and protection of sensitive and regulated information in a shared environment.



Cloud Security - Denial of Service (DoS)

1. What is a DoS Attack?

- A denial-of-service (DoS) attack is a cyber attack aiming to render a computer or device unavailable to its intended users by interrupting its normal functioning.
- The attack overwhelms the targeted machine with requests, making it unable to process normal traffic, thus denying service to additional users.
- DoS attacks are typically launched using a single computer.

2. DoS Attack Categories:

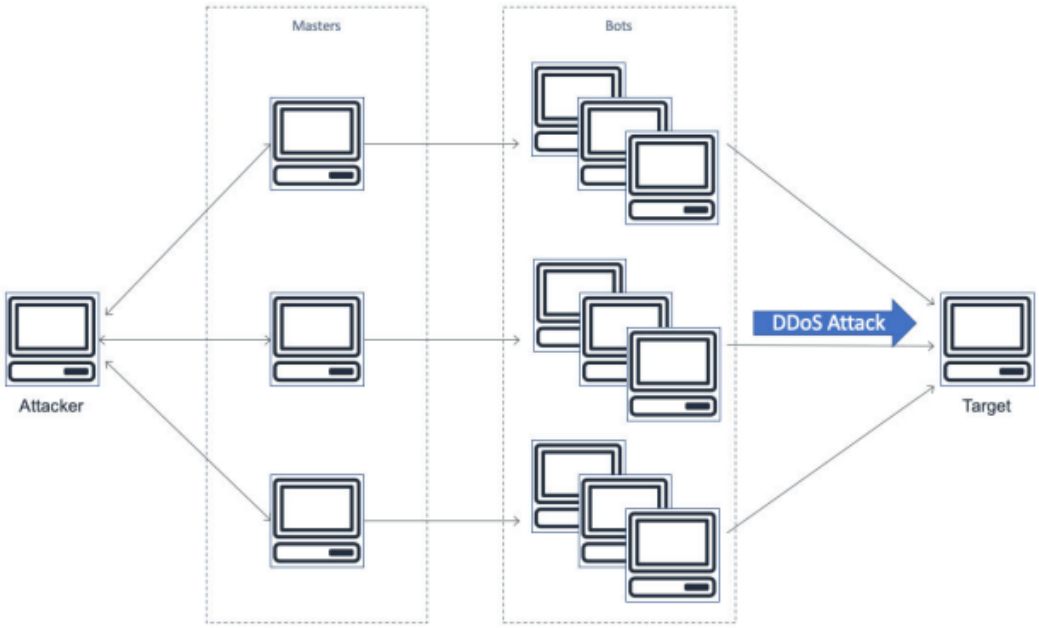
- **Buffer Overflow Attacks:**
  - Involves a memory buffer overflow causing a machine to consume all available hard disk space, memory, or CPU time.
  - Results in sluggish behavior, system crashes, or other server issues, leading to denial-of-service.
- **Flood Attacks:**
  - Involves saturating a targeted server with an overwhelming amount of packets, exceeding its capacity.
  - Results in denial-of-service by overloading the server.

3. Indicators of a DoS Attack:

- Slow network performance, such as long load times for files or websites.
- Inability to load a specific website or web property.
- Sudden loss of connectivity across devices on the same network.

Cloud Security - Distributed Denial of Service (DDoS)

1. What is a DDoS Attack?



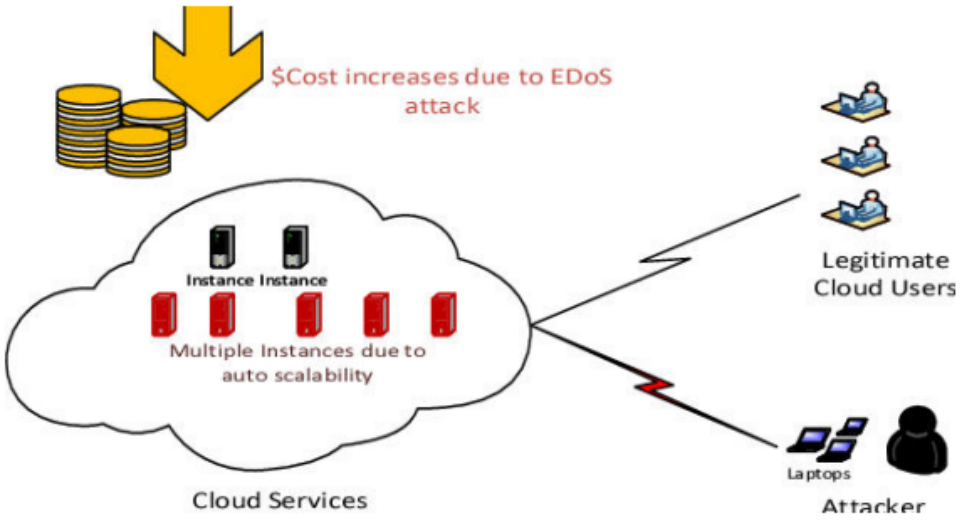
- A distributed denial-of-service (DDoS) attack is a malicious attempt to disrupt the normal traffic of a targeted server, service, or network by overwhelming it with a flood of Internet traffic.
- DDoS attacks utilize multiple compromised computer systems as sources of attack traffic, forming a network of Internet-connected machines.
- These compromised machines, including computers and IoT devices, are controlled remotely by the attacker and form a botnet.
- The attacker sends remote instructions to each bot, directing them to send requests to the target's IP address, overwhelming the server or network and causing denial-of-service to legitimate traffic.

2. DDoS Attack Characteristics:

- **Attack Mechanism:**
  - DDoS attacks are carried out using networks of compromised machines (botnets) controlled remotely by the attacker.
- **Botnets:**
  - Botnets consist of infected computers and IoT devices, referred to as bots or zombies.
  - Each bot sends requests to the target's IP address, overwhelming its server or network.
- **Difficulty in Differentiating Traffic:**
  - Identifying and separating attack traffic from normal traffic is challenging as each bot is a legitimate Internet device.

3. DDoS vs. EDoS:

- **DDoS (Distributed Denial of Service):**
  - Aims to cease all services provided by the service provider, interrupting maximum server resources in a short interval.
- **EDoS (Economic Denial of Sustainability):**
  - Targets an individual by gradually pushing illegal data toward the genuine user over a longer duration, maximizing the financial costs of cloud-based consumers.
  - EDoS focuses on maximizing the financial costs of cloud-based consumers through cloud-based resources, affecting cloud computations in terms of performance and price models.



DDoS attack	EDoS attack
Degrade/block cloud services	Makes cloud resource economically unsustainable
The attacker period is very short	The attacker period is long
The attacker occurs above the EDoS attack zone	The attacks appear between normal data traffic and DDoS attack region