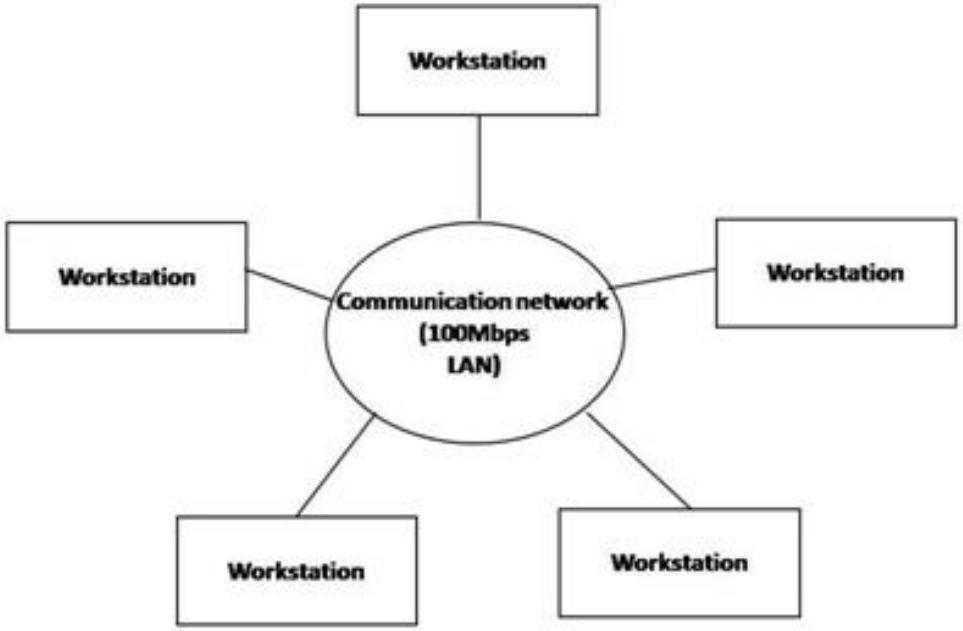
CLOUD COMPUTING

Master-slave v/s p2p models

Prof.S.Suganthi
Department of Computer Science and Engineering

Distributed Systems

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



Two main architectures:

Master-Slave architecture

Roles of entities are *asymmetric*

Peer-to-Peer architecture

Roles of entities are *symmetric*

Master-Slave Architecture

A master-slave architecture can be characterized as follows:

1) Nodes are ***unequal*** (there is a hierarchy)
   - Vulnerable to *Single-Point-of-Failure* (SPOF)

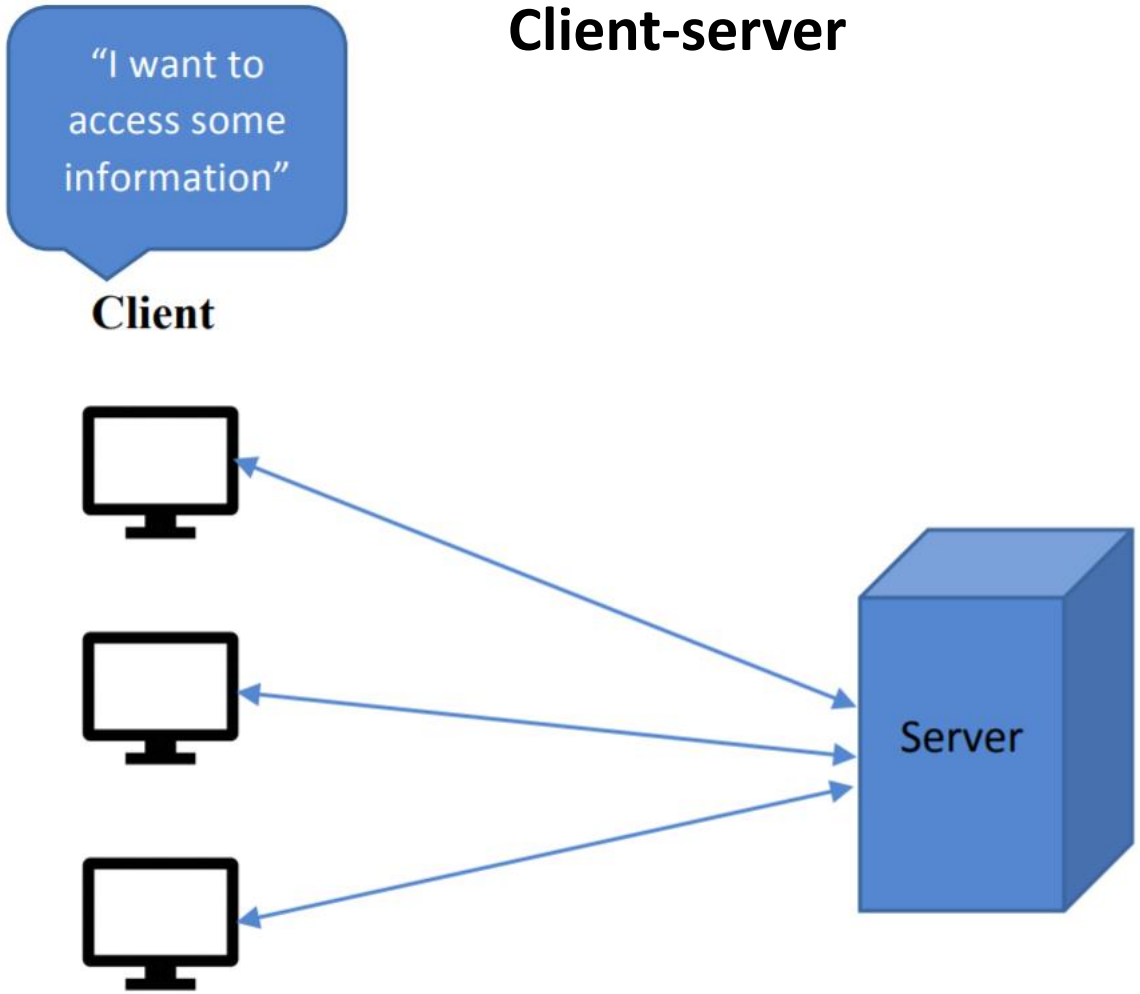
2) The master acts as a ***central coordinator***
   - Decision making becomes easy

3) The underlying system ***cannot scale out*** indefinitely
   - The master can render a *performance bottleneck* as the number of workers is increased

CLOUD COMPUTING

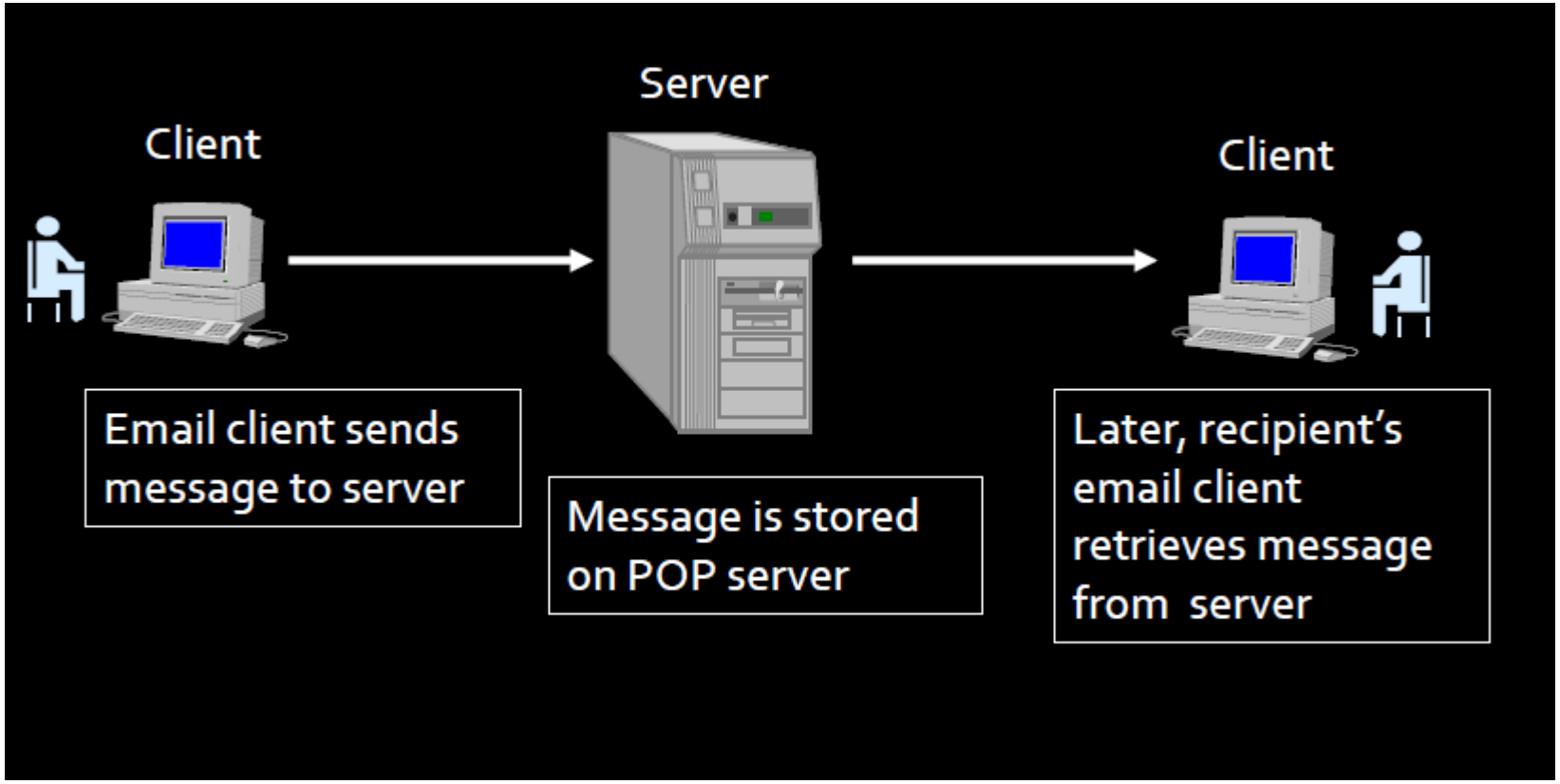
Master-Slave Architecture

Client-server

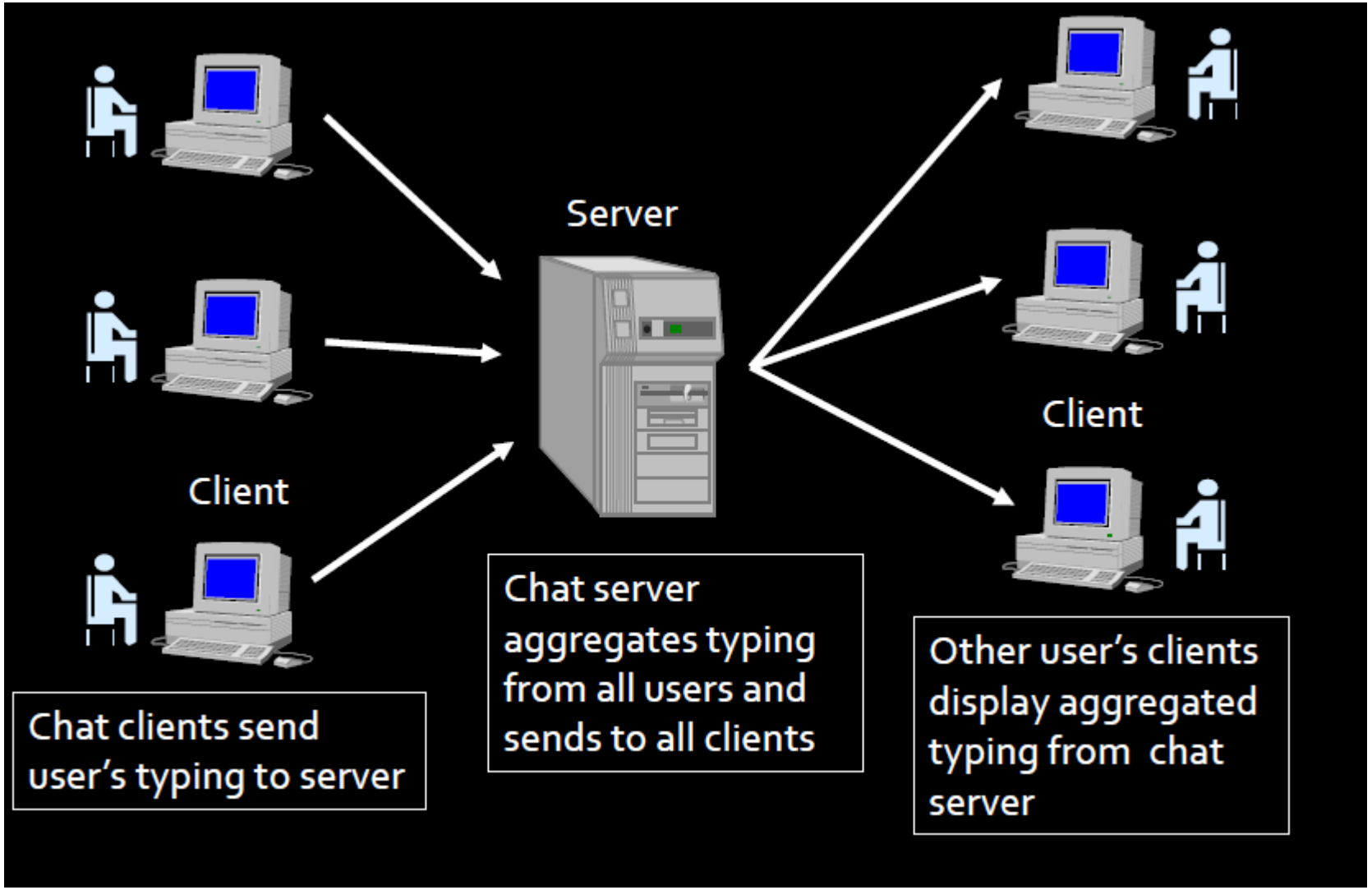


CLOUD COMPUTING

Examples - Email Application

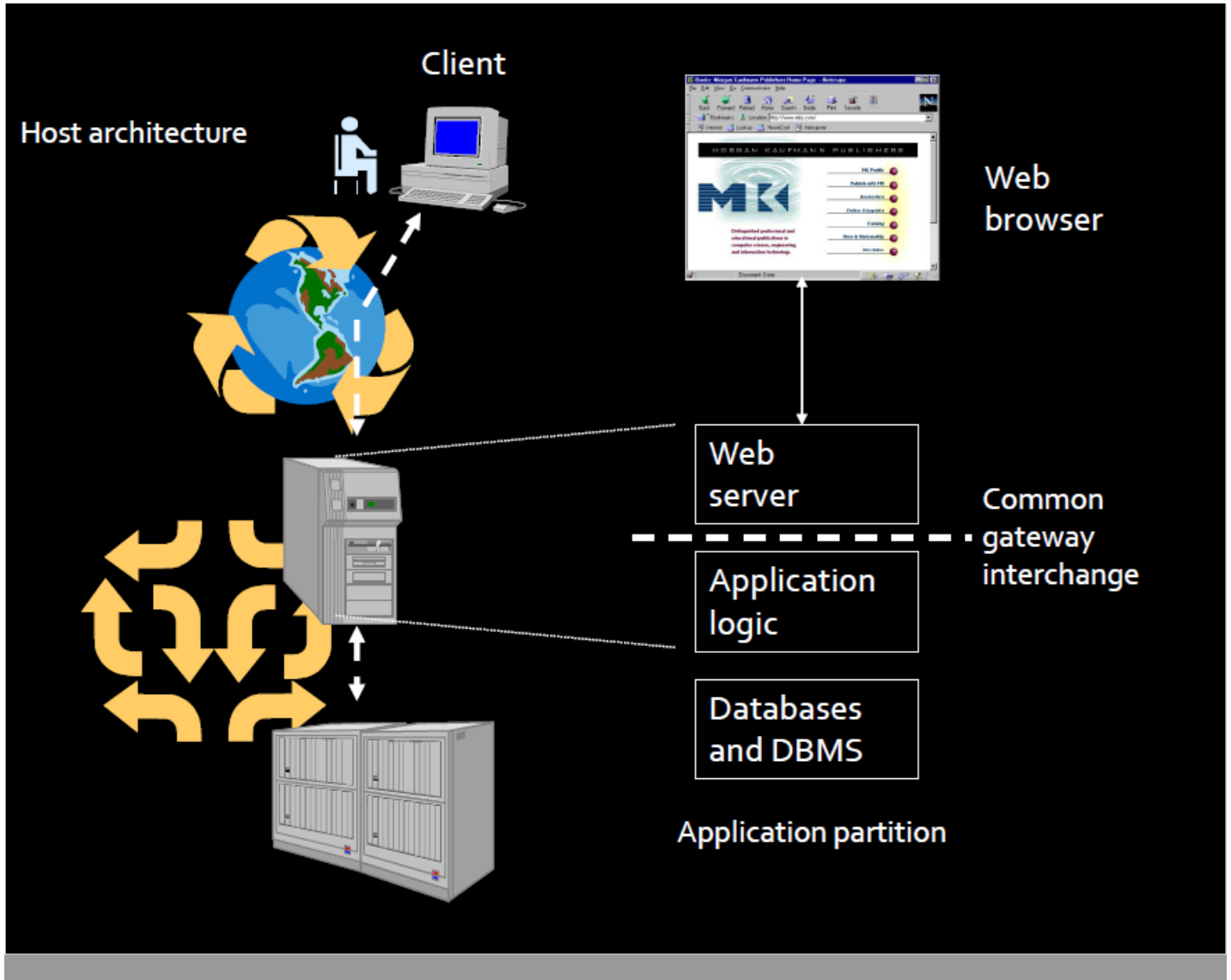


Examples - Chat Application



CLOUD COMPUTING

3 tier Client-Server Architecture

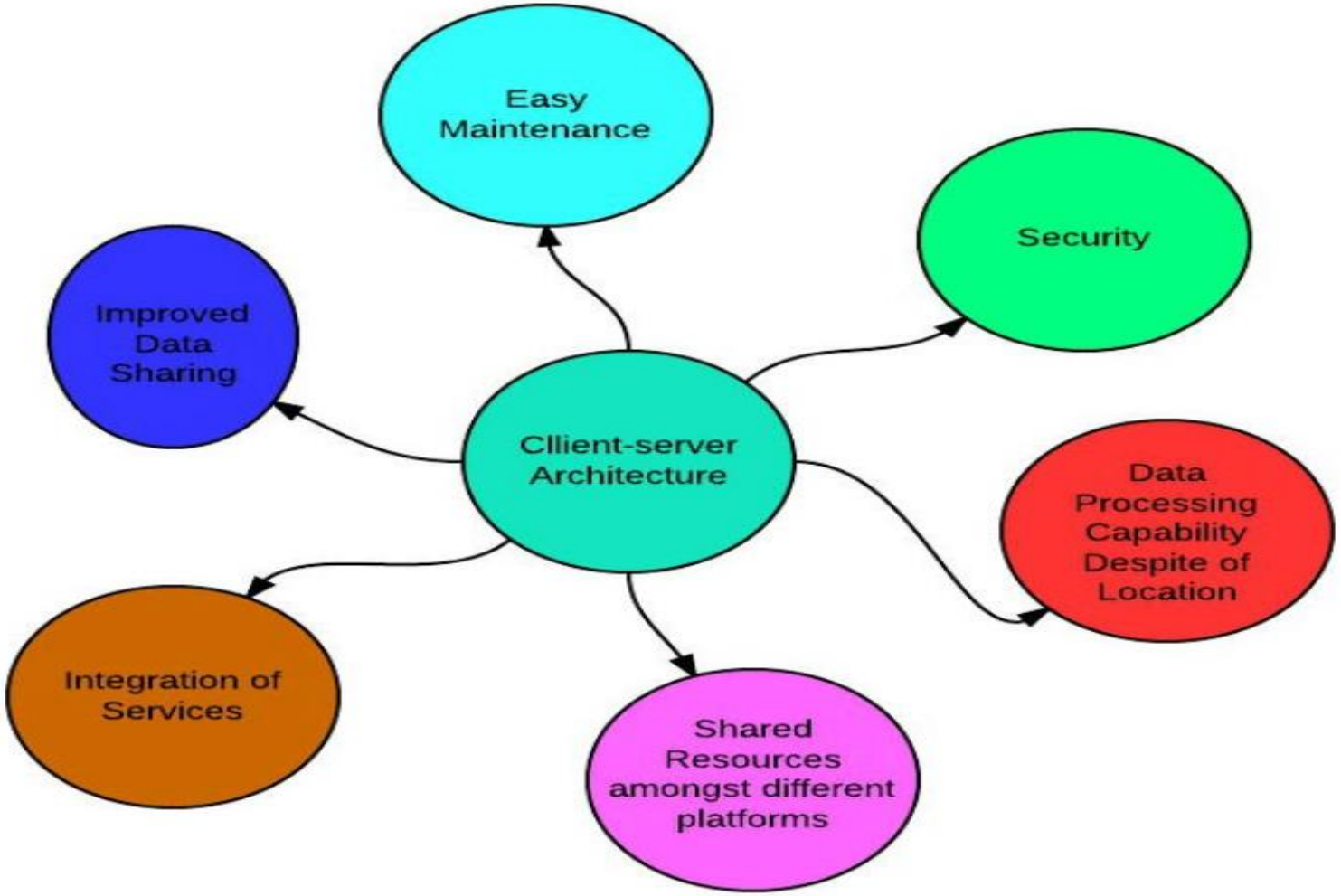


Information about the client is stored in a middle tier rather than on the client to simplify application deployment. This architecture model is most common for web applications.

CLOUD COMPUTING

Client-Server Architecture

- Advantages



Source: https://sites.google.com/site/clientserverarchitecture/advantages-of-client-server-architecture

- **Improved Data Sharing:**

Data is retained by usual business processes and manipulated on a server is available for designated users (clients) over an authorized access.

- **Integration of Services:**

Every client is given the opportunity to access corporate information via desktop interface eliminating the necessity to log into a terminal mode or processor.

- **Shared Resources Amongst Different Platforms:**

Application used for client-server model is built regardless of the hardware platform or technical background of the entitled software (operating system software) providing an open computing environment, enforcing users to obtain the services of clients and servers (database, application and communication services)

- **Data Processing Capability Despite the Location:**

Client-server users can directly log into a system despite of the location or technology of the processors.

- **Easy Maintenance:**

Client-server architecture is distributed model representing dispersed responsibilities among independent computers integrated across a network. Therefore, it's easy to replace, repair, upgrade and relocate a server while client remains unaffected. This unaware change is called as Encapsulation.

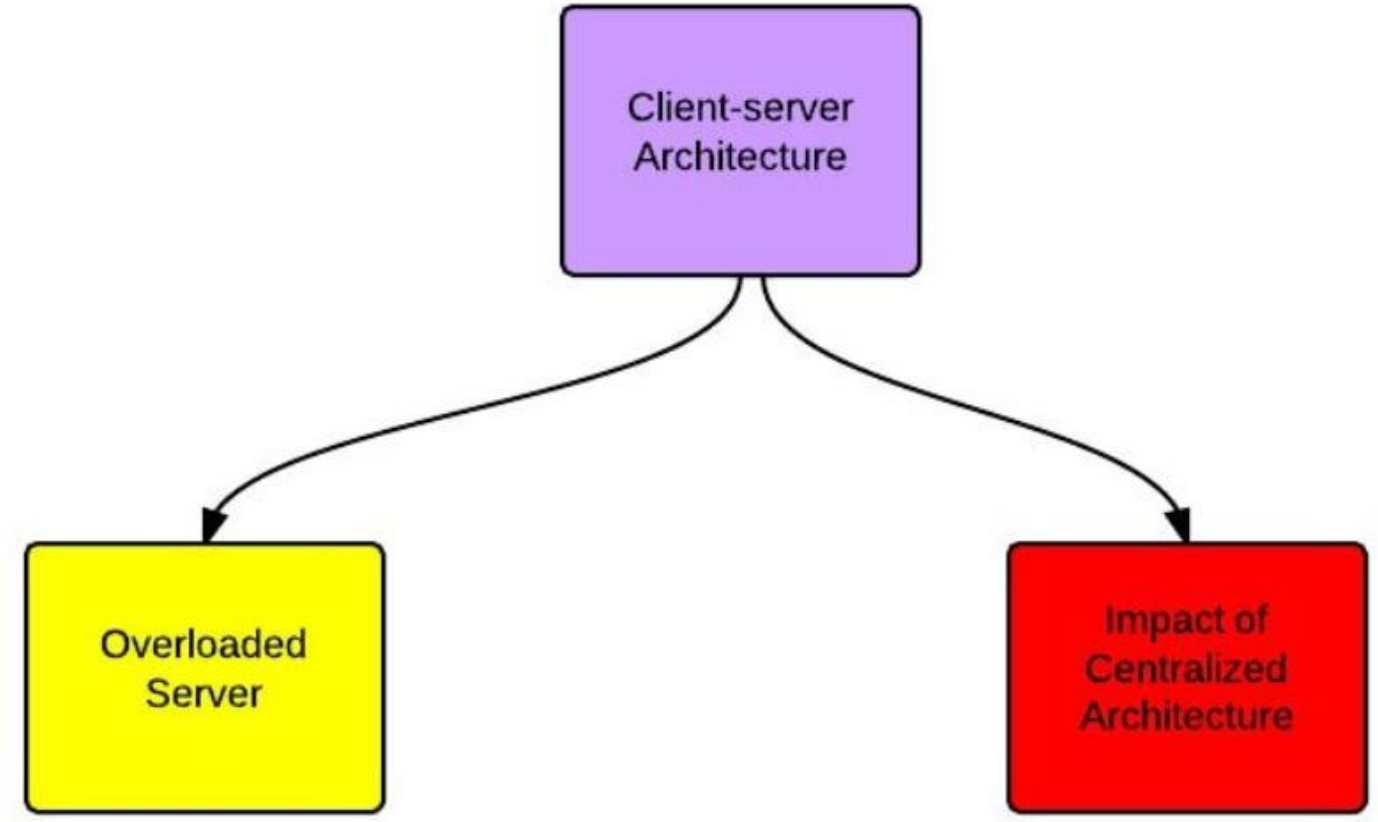
- **Security:**

Servers have better control access and resources to ensure that only authorized clients can access or manipulate data and server updates are administered effectively.

CLOUD COMPUTING

Client-Server Architecture

- Disadvantages



Source: https://sites.google.com/site/clientserverarchitecture/advantages-of-client-server-architecture

- **Overloaded Servers:**

When there are frequent simultaneous client requests, server severely get overloaded, forming traffic congestion.

- **Impact of Centralized Architecture:**

Since it is centralized, if a critical server failed, client requests are not accomplished. Therefore, client-server lacks the robustness of a good network.

What is Peer-to-Peer?

- A model of communication where every node in the network acts alike.
- As opposed to the Client-Server model, where one node provides services and other nodes use the services.
- In a P2P system, every node acts as both a client and a server, providing part of the system resources.
- All client machines act autonomously to join or leave the system freely. This implies that no master-slave relationship exists among the peers. No central coordination or central database is needed.

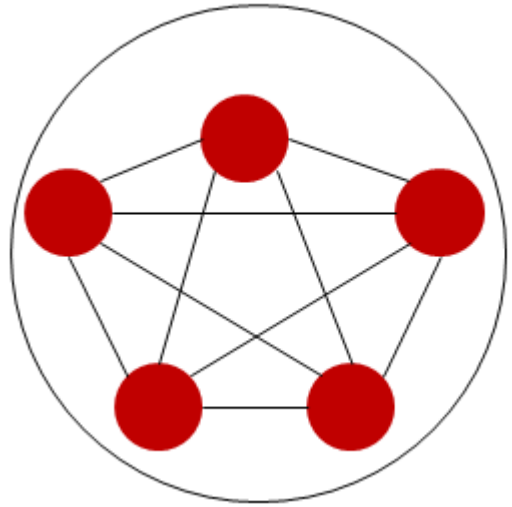
P2P Architecture

- A peer-to-peer (P2P) architecture can be characterized as follows:

1) All nodes are equal (no hierarchy)
   - No Single-Point-of-Failure (SPOF)

2) A central coordinator is not needed
   - But, decision making becomes harder

3) The underlying system can scale out indefinitely
   - In principle, no performance bottleneck



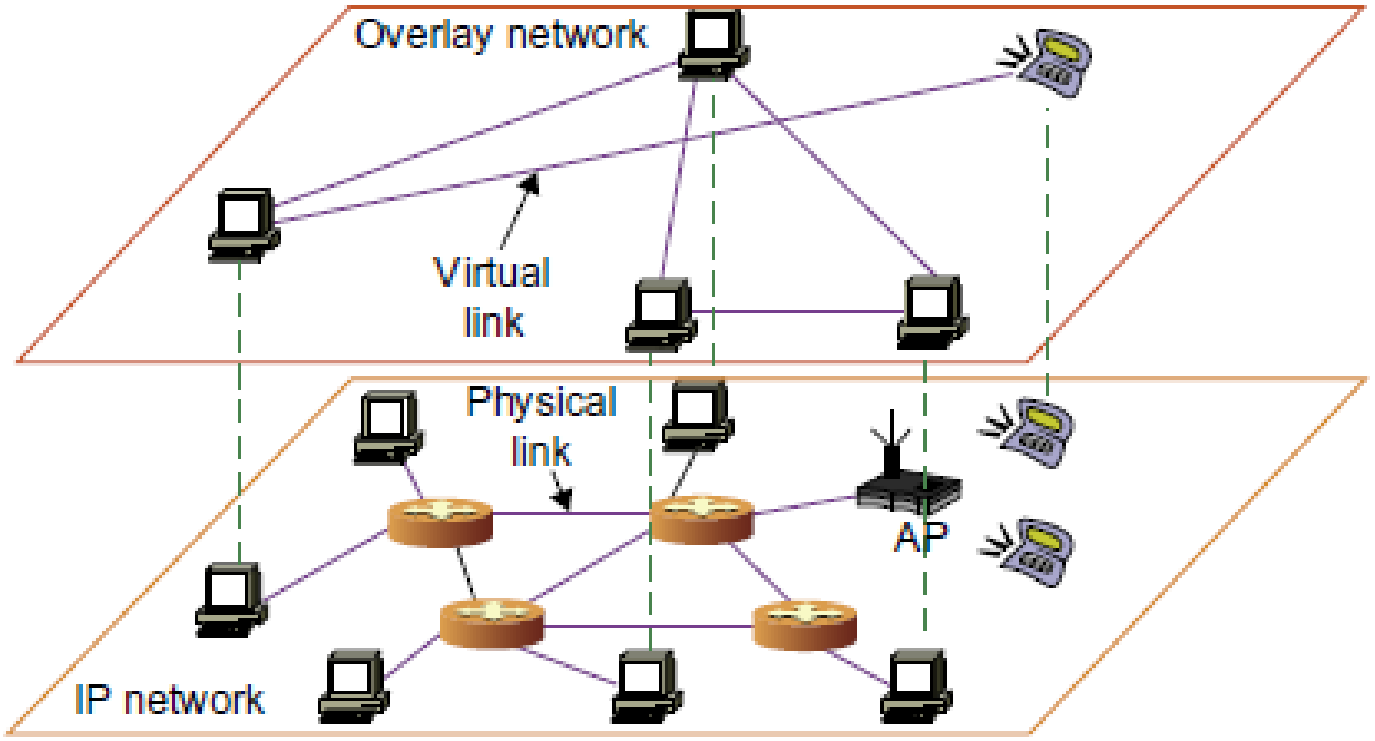
P2P Architecture

4) 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)

5) 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

CLOUD COMPUTING

P2P Architecture



The structure of a P2P system by mapping a physical IP network to an overlay network built with virtual links.

Advantages of P2P Computing

- **No central point of failure**
  - E.g., the Internet and the Web do not have a central point of failure.
  - Most internet and web services use the client-server model (e.g. HTTP), so a specific service does have a central point of failure.
- **Scalability**
  - Since every peer is alike, it is possible to add more peers to the system and scale to larger networks.

Disadvantages of P2P Computing

- **Decentralized coordination**
  - How to keep global state consistent?
  - Need for distributed coherency protocols.
- **All nodes are not created equal.**
  - Computing power, bandwidth have an impact on overall performance.
- **Programmability**
  - As a corollary of decentralized coordination.

P2P Computing Applications

- File sharing
- Process sharing
- Collaborative environments

P2P File Sharing Applications

- **Peer-to-peer file sharing** is the distribution and sharing of digital media using peer-to-peer (P2P) networking technology.
- P2P file sharing allows users to access media files such as books, music, movies, and games using a P2P software program that searches for other connected computers on a P2P network to locate the desired content.
- The nodes (peers) of such networks are end-user computers and distribution servers (not required).

P2P File Sharing Applications

- *Several factors contributed to the widespread adoption and facilitation of peer-to-peer file sharing*.
  - increasing Internet bandwidth,
  - the widespread digitization of physical media, and
  - the increasing capabilities of residential personal computers.
  - Users are able to transfer one or more files from one computer to another across the Internet through various file transfer systems and other file-sharing networks.
  - E.g., Napster, Gnutella, Freenet, KaZaA (FastTrack) etc

P2P Process Sharing Applications

- For large-scale computations
- Data analysis, data mining, scientific computing
- E.g., SETI@Home, Folding@Home, distributed.net, World-Wide Computer

CLOUD COMPUTING

P2P Colloborative Applications

- For remote real-time human collaboration.
- Instant messaging, virtual meetings, shared whiteboards, teleconferencing, tele- presence.
- E.g., AOL Messenger, Yahoo! Messenger, Jabber, MS Netmeeting, NCSA Habanero, Games

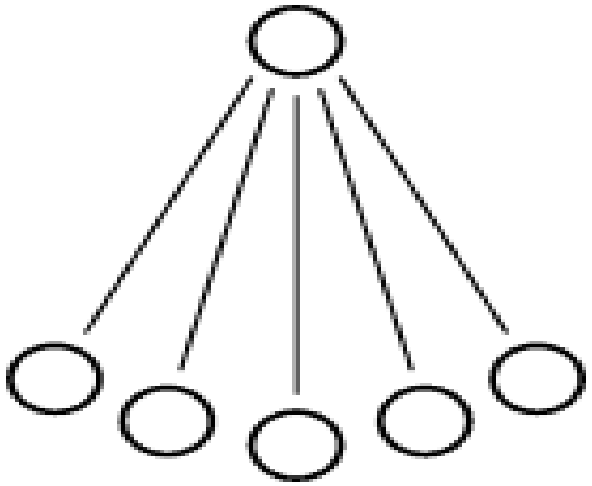
CLOUD COMPUTING

P2P Topologies

- Centralized
- Ring
- Hierarchical
- Decentralized
- Hybrid

P2P Topologies: Centralized Topology

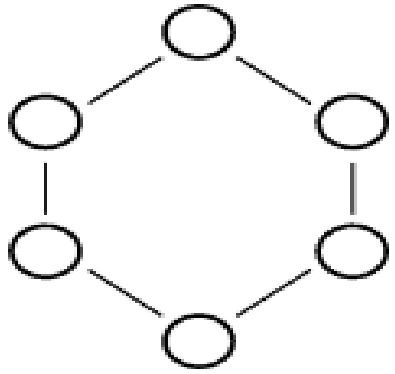
- Centralized



Manageable	✓	System is all in one place
Coherent	✓	All information is in one place
Extensible	X	No one can add on to system
Fault Tolerant	X	Single point of failure
Secure	✓	Simply secure one host
Lawsuit-proof	X	Easy to shut down
Scalable	?	One machine. But in practice?

P2P Topologies: Ring Topology

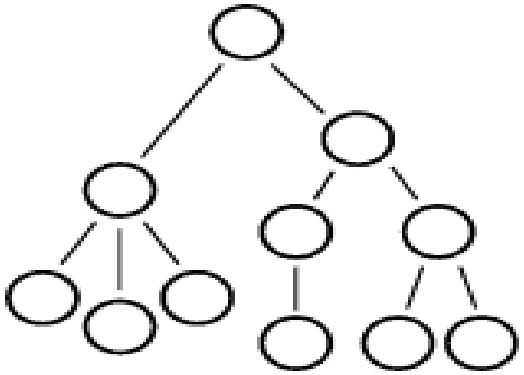
- Ring



Manageable	✓	Simple rules for relationships
Coherent	✓	Easy logic for state
Extensible	X	Only ring owner can add
Fault Tolerant	✓	Fail-over to next host
Secure	✓	As long as ring has one owner
Lawsuit-proof	X	Shut down owner
Scalable	✓	Just add more hosts

P2P Topologies: Hierarchical Topology

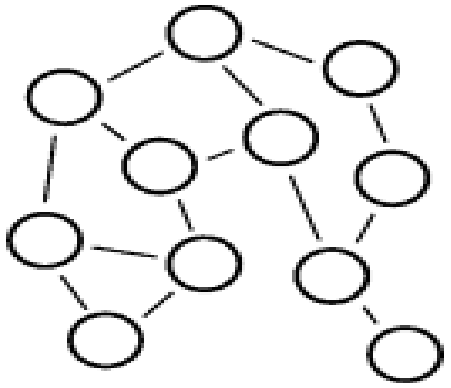
- Hierarchical



Manageable	½ Chain of authority
Coherent	½ Cache consistency
Extensible	½ Add more leaves, rebalance
Fault Tolerant	½ Root is vulnerable
Secure	X Too easy to spoof links
Lawsuit-proof	X Just shut down the root
Scalable	✓ Hugely scalable – DNS

P2P Topologies: Decentralized Topology

- Decentralized



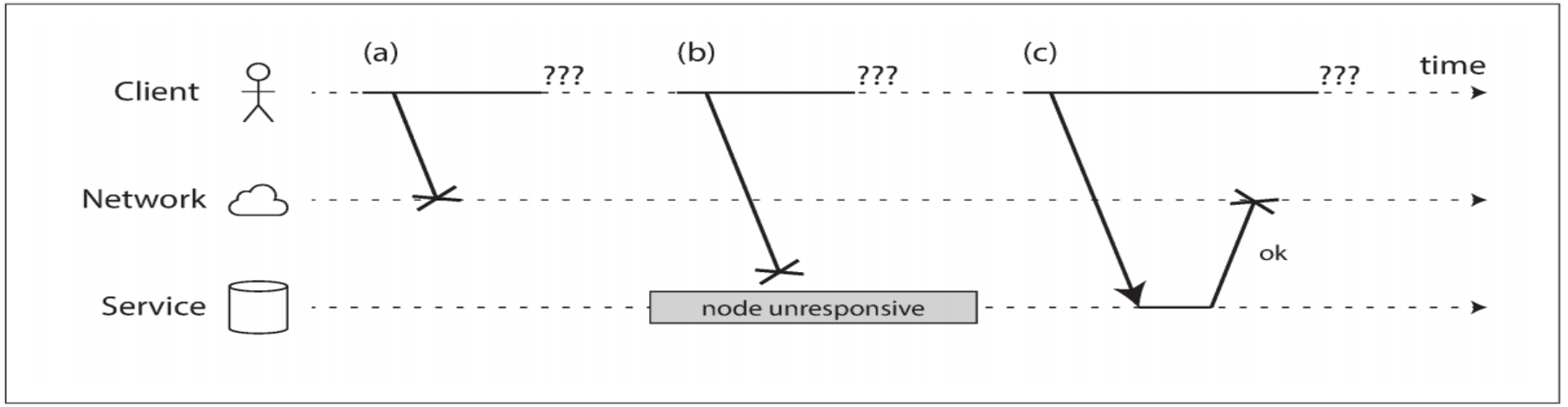
Manageable	X	Very difficult, many owners
Coherent	X	Difficult, unreliable peers
Extensible	✓	Anyone can join in!
Fault Tolerant	✓	Redundancy
Secure	X	Difficult, open research
Lawsuit-proof	✓	No one to sue
Scalable	?	Theory – yes : Practice – no

CLOUD COMPUTING

Trouble with Distributed Systems

If you send a request and expect a response, 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
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 later



Ref: T3: Chapter 8: Trouble with Distributed Sytems

CLOUD COMPUTING

Trouble with Distributed Systems

- In a distributed system,
  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. This is known as a ***partial failure*.**
- The difficulty is that partial failures are ***nondeterministic***: if you try to do anything involving multiple nodes and the network, it may sometimes work and sometimes unpredictably fail.
- 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 nondeterministic!
- This nondeterminism and possibility of partial failures is what makes distributed systems hard to work with
- If we want to make distributed systems work, accept the possibility of partial failure and build fault-tolerance mechanisms into the software.
- ***In other words, build a reliable system from unreliable components***

Ref: T3: Chapter 8: Trouble with Distributed Sytems

What do you mean by Reliability?

- The application performs the function that the user expected.
- It can tolerate the user making mistakes or using the software in unexpected ways.
- Its performance is good enough for the required use case, under the expected load and data volume.
- The system prevents any unauthorized access and abuse.
- If all those things together mean "working correctly," then we can understand *reliability* as meaning, roughly, "***continuing to work correctly, even when things go wrong***."
- The things that can go wrong are called *faults,* and systems that anticipate faults and can cope with them are called ***fault-tolerant* or *resilient*.**

Ref: T3: Chapter 8: Trouble with Distributed Sytems

Reliability

- *Reliability refers to the probability that the system will meet certain performance standards in yielding correct output for a desired time duration.*
- High reliability is extremely important in critical applications. There are many applications hosted on the cloud for whom failure or an outage of service can mean the loss of lives.
- A common metric to measure reliability is the ***Mean Time Between Failures (MTBF)***

  average length of operating time between one failure and the next

  MTBF = Total uptime / # of Breakdowns

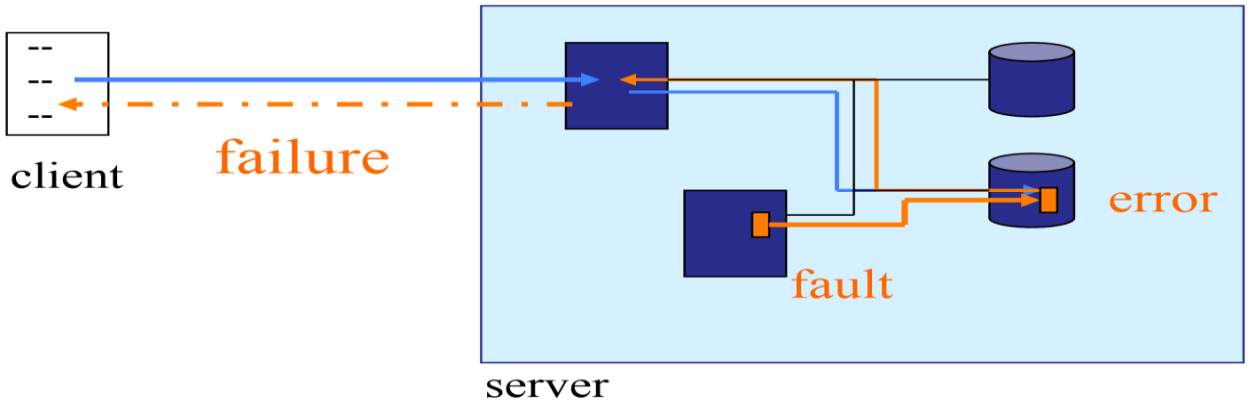
  For example, if the specified operating time is 24 hours and in that time 12 failures occur, then the MTBF is two hours.
- Reliability can be used to understand how well the service will be available in context of different real-world conditions.

https://www.bmc.com/blogs/reliability-vs-availability

Failure

What is failure?

- A system is said to “fail” when it cannot meet its requirements.
- A failure is brought about by the existence of errors in the system. The cause of an error is called a fault.
- Faults can be classified as
  - **Transient** — Appears once, then disappears.(software bugs etc)
  - **Intermittent** — occurs, vanishes, reappears; but: follows no real pattern(loose connection.)
  - **Permanent** — once it occurs, only the replacement/repair of a faulty component will allow the service to function normally.(disk-head crash)



CLOUD COMPUTING

Types of Failure in Designer perspective

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

Crash: fail-stop, fail-safe (*detectable*), fail-silent (*seems to have crashed*)

Types of Failure in Designer perspective

A ***fail-silent fault (***fail-stop***)***is one where the faulty unit stops functioning and produces no ill output (it produces no output or produces output to indicate failure).

A ***Byzantine fault*** is one where the faulty unit continues to run but produces incorrect results. Byzantine faults are obviously more troublesome to deal with.

Fault Vs Failure

- A ***fault*** is usually defined as one component of the system deviating from its spec, whereas a ***failure*** is when the system as a whole stops providing the required service to the user.
- 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

Ref: T3: Chapter 8: Trouble with Distributed Sytems

Timeouts and Unbounded Delay: how long should the time-out be?

- Every packet is either delivered within some time *d*, or it is lost, but delivery never takes longer than *d*
- Guarantee that a non-failed node always handles a request within some time *r*
- In this case, you could guarantee that every successful request receives a response within time 2*d* + *r* and
- 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***

They try to deliver packets as quickly as possible, but there is no upper limit on the time it may take for a packet to arrive, and most server implementations cannot guarantee that they can handle requests within some maximum time

Network Congestion and Queueing

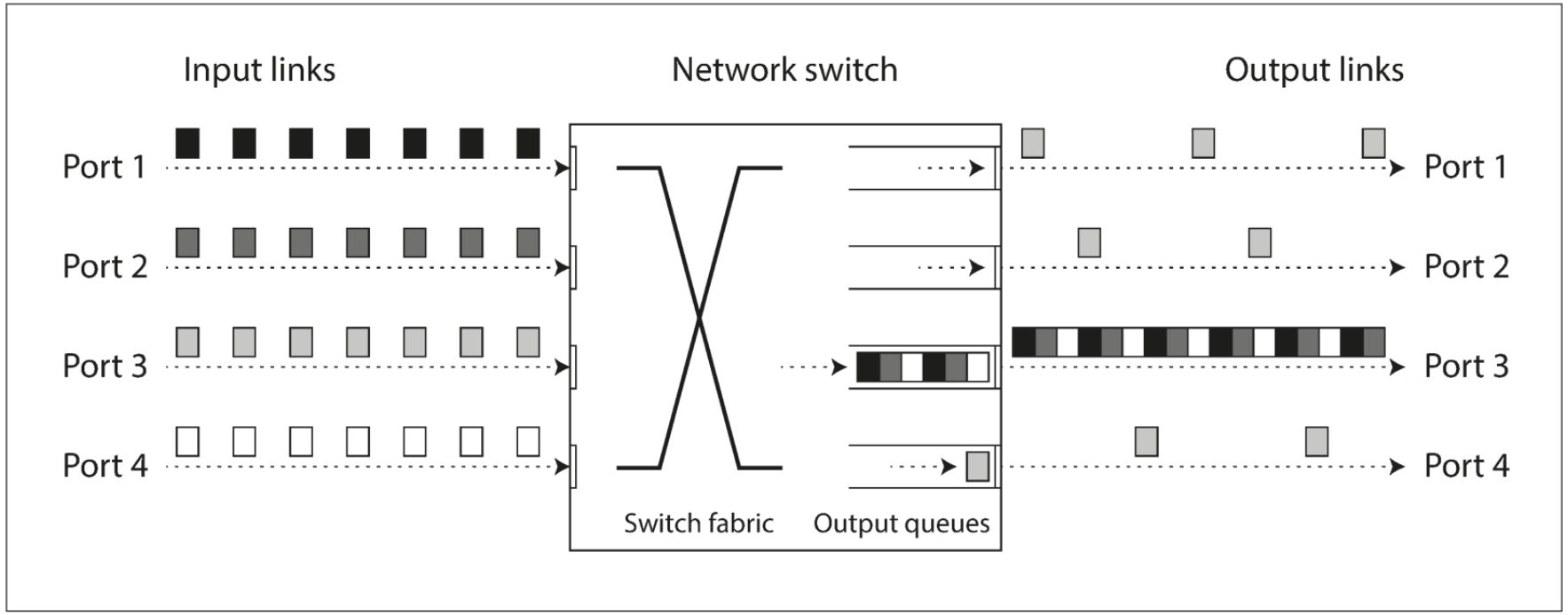


Figure 8-2. If several machines send network traffic to the same destination, its switch queue can fill up. Here, ports 1, 2, and 4 are all trying to send packets to port 3.

Network Congestion and Queueing

- The variability of packet delays on computer networks is most often due to queueing
- 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, the packet is dropped, so it needs to be resent—even though the network is functioning fine.
- In *virtualized environments*, a running operating system is often paused for tens of milliseconds while another virtual machine uses a CPU core.
- During this time, the VM cannot consume any data from the network, so the incoming data is queued (buffered) by the virtual machine monitor
- 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

Fault Detection

- In *public clouds and multi-tenant datacenters*, resources are shared among many customers:
- the network links and switches, and even each machine's network interface and CPUs (when running on virtual machines), are shared.
- Batch workloads such as MapReduce can easily saturate network links.
- As you have 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

Fault Detection

Choose timeouts experimentally:

- 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 of the application's characteristics, determine an appropriate trade-off between failure detection delay and 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 time-outs according to the observed response time distribution.
- Phi Accrual failure detector, which is used in Akka and Cassandra, TCP retransmission timeouts also work similarly

Synchronous Vs Asynchronous Network

- if we could rely on the network to deliver packets with some fixed maximum delay, and not to drop packets, Why can't we solve this at the hardware level and make the network reliable so that the software doesn't need to worry about it?
- CircuitSwitching -Synchronous-does not suffer from queueing because there is no queueing, the maximum end-to-end latency of the network is fixed which is termed as *bounded delay*
- Can we not simply make network delays predictable?
- Circuit in a telephone network is very different from a TCP connection

Fault Detection

- Datacenter networks and the internet use packet switching as they are optimized for *bursty traffic* which suffer from queueing and thus unbounded delays in the network
- Latency guarantees are achievable in certain environments, if resources are statically partitioned (e.g., dedicated hardware and exclusive bandwidth allocations). However, it comes at the cost of reduced utilization—in other words, it is more expensive.
- On the other hand, multi-tenancy with dynamic resource partitioning provides better utilization, so it is cheaper, but it has the downside of variable delays.
- Variable delays in networks are not a law of nature, but simply the result of a cost/benefit trade-off

Fault Detection

- Currently deployed technology does not allow us to make any guarantees about delays or reliability of the network
- Assume that *network congestion, queueing, and unbounded delays will happen.*
- Consequently, *there's no "correct" value for timeouts—they need to be determined experimentally*

Three Common Failure Models

Crash-stop faults

- *A*n algorithm assume that a node can fail in only one way, namely by crashing.
- This means that the node may suddenly stop responding at any moment, and thereafter that node is gone forever it never comes back.

Crash-recovery faults

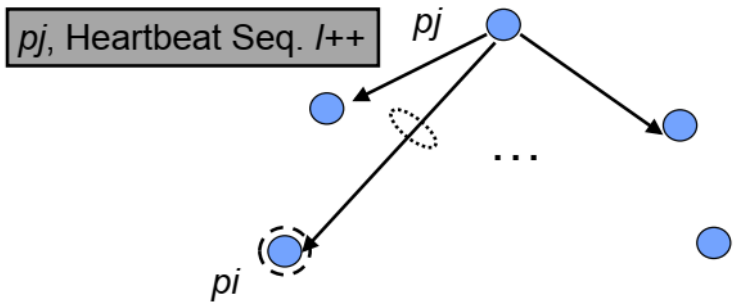
- *N*odes may crash at any moment, and perhaps start responding again after some unknown time.
- In the crash-recovery model, nodes are assumed to have stable storage (i.e., nonvolatile disk storage) that is preserved across crashes, while the in-memory state is assumed to be lost.

Byzantine (arbitrary) faults

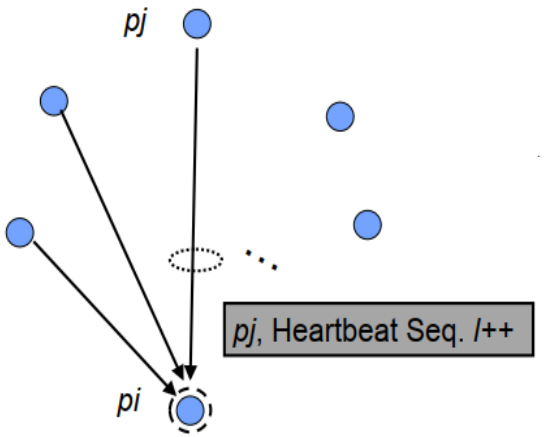
- Nodes may do absolutely anything, including trying to trick and deceive other nodes

Failure Detection in Distributed System

All-to-All Heartbeat

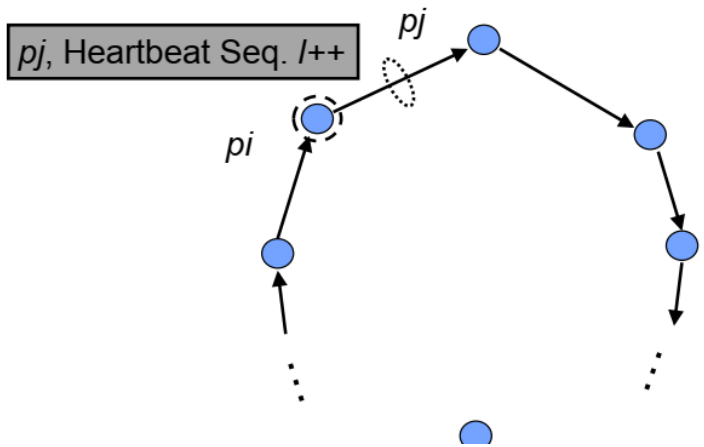


Centralized Heartbeat



Advantage: Everyone is able to keep track of everyone

Ring Heartbeat



Byzantine Fault Tolerant

- A system is *Byzantine fault-tolerant if it continues to operate correctly even if some of the nodes are malfunctioning and not obeying the protocol, or if malicious attackers are interfering with the network.*

Byzantine agreement:

- Must find a way of identifying faulty nodes and corrupted/forged messages
- *General strategy*
    - everyone communicate not only their own info, but everything they've heard from everyone else. This allows catching lying, cheating nodes.
    - If there is a majority that behaves consistently with each other, can reach consensus

Failover Architecture

- **Failover** is *switching to 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*

https://networksandservers.blogspot.com/2011/09/failover-clustering-iv.html

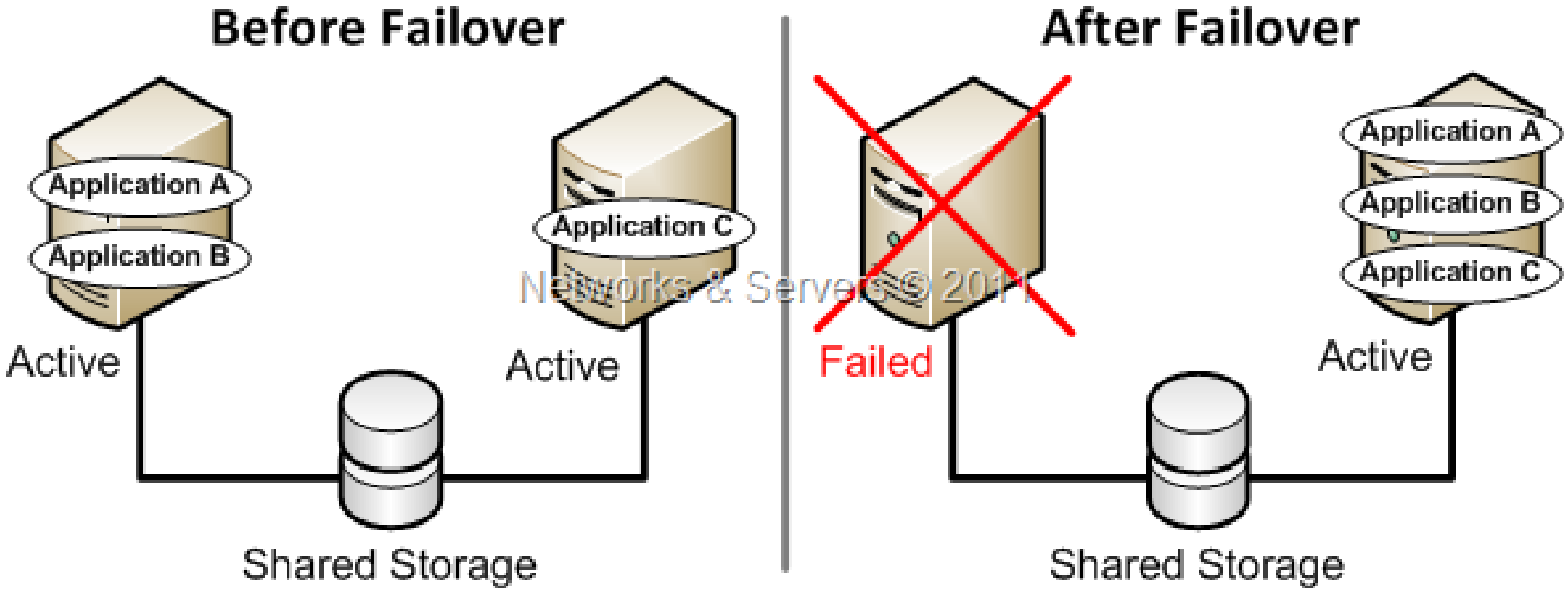
Failover Architecture

Active-Active Failover Architecture(sometimes referred to as 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.

Failover Architecture

Active-Active Failover Architecture



Failover Architecture

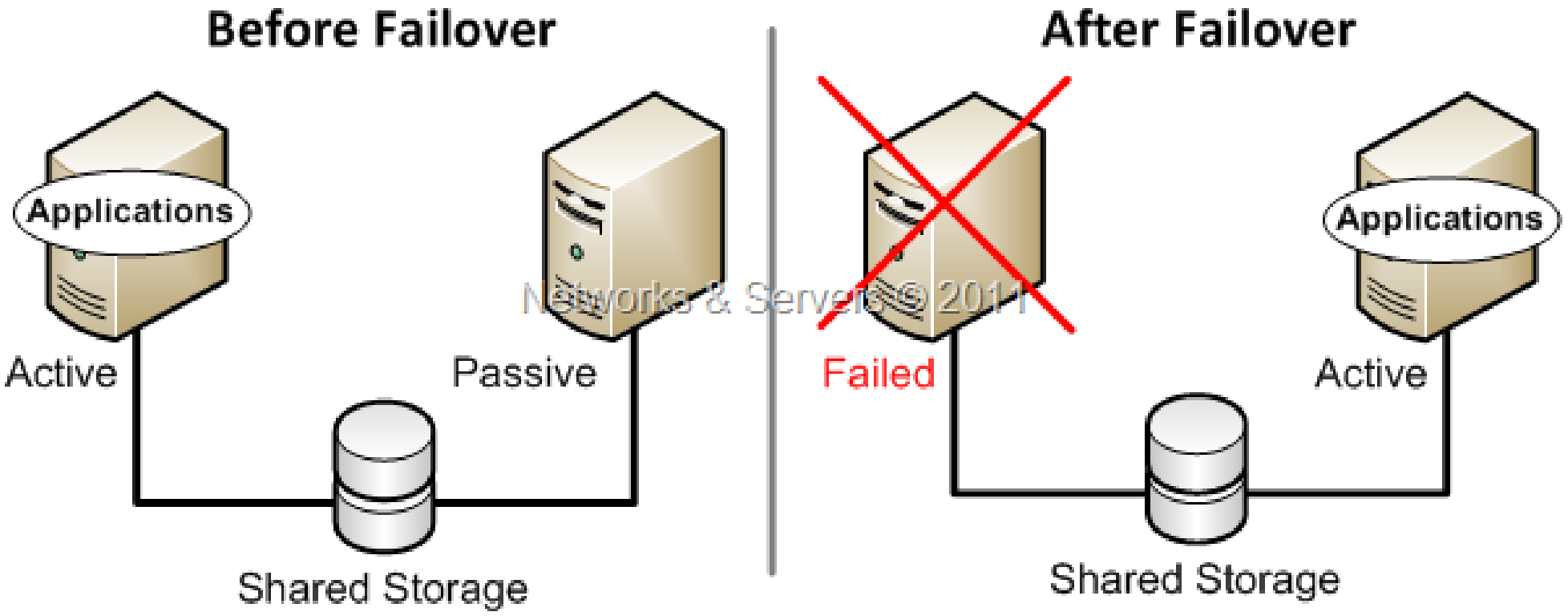
Active-Passive Failover Architecture (or asymmetric)

- 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*

Failover Architecture

Active-Passive Failover Architecture



Key aspect of Distributed Computing System

Performance

- In addition, system availability and application flexibility are two other important design goals

Availability

- used to describe the period of time when a service is available, as well as the time required by a system to respond to a request made by a user.
- often expressed as a percentage indicating how much uptime is expected from a particular system or component in a given period of time, where a value of 100% would indicate that the system never fails.

System Availability

Uptime

- refers to the time-period during which a computer is running or is operational.
- standard of measurement calculated in terms of a percentile.
- standard is 99.999 percentile also called five 9s.

Downtime

- time-period for which the computer is unavailable or non-operational.
- also called *outage duration.*
- The unavailability of the system or network may be due to any system failures (unplanned event) which may occur from some kind of a software crash or from communication failures (network outages).
- Maintenance routines (this are planned events) can also lead to unavailability of the system and thereby causing Downtime.

System Availability

Uptime and Downtime

- critical for determining the success level of the real-time services or systems.
- help us quantify the success value of these services or systems.
- Generally, every service level agreements and contracts include an uptime/downtime ratio that shows for how long that service will be operational and available to the users.

Let us consider a website was monitored for 24 hours (= 86400 seconds). Now in this timeframe say, the monitor was down for about 10 minutes (=600 seconds).

Downtime

600/86400=0.0069 = 0.69%

Uptime

100% — Downtime %

= 100% — 0.69% = 99.31%

https://gaurav-cnwn1.medium.com/all-you-need-to-know-about-uptime-downtime-and-runtime-errors-51e9ebca1a69

CLOUD COMPUTING

System Availability

- *Any failure that will pull down the operation of the entire system is called a* ***single point of failure***. The rule of thumb is to design a dependable computing system with no single point of failure.
- In general, as a distributed system increases in size, availability decreases due to a higher chance of failure and a difficulty in isolating the failures.
- Most clusters are designed to have High Availability with ***failover capability***

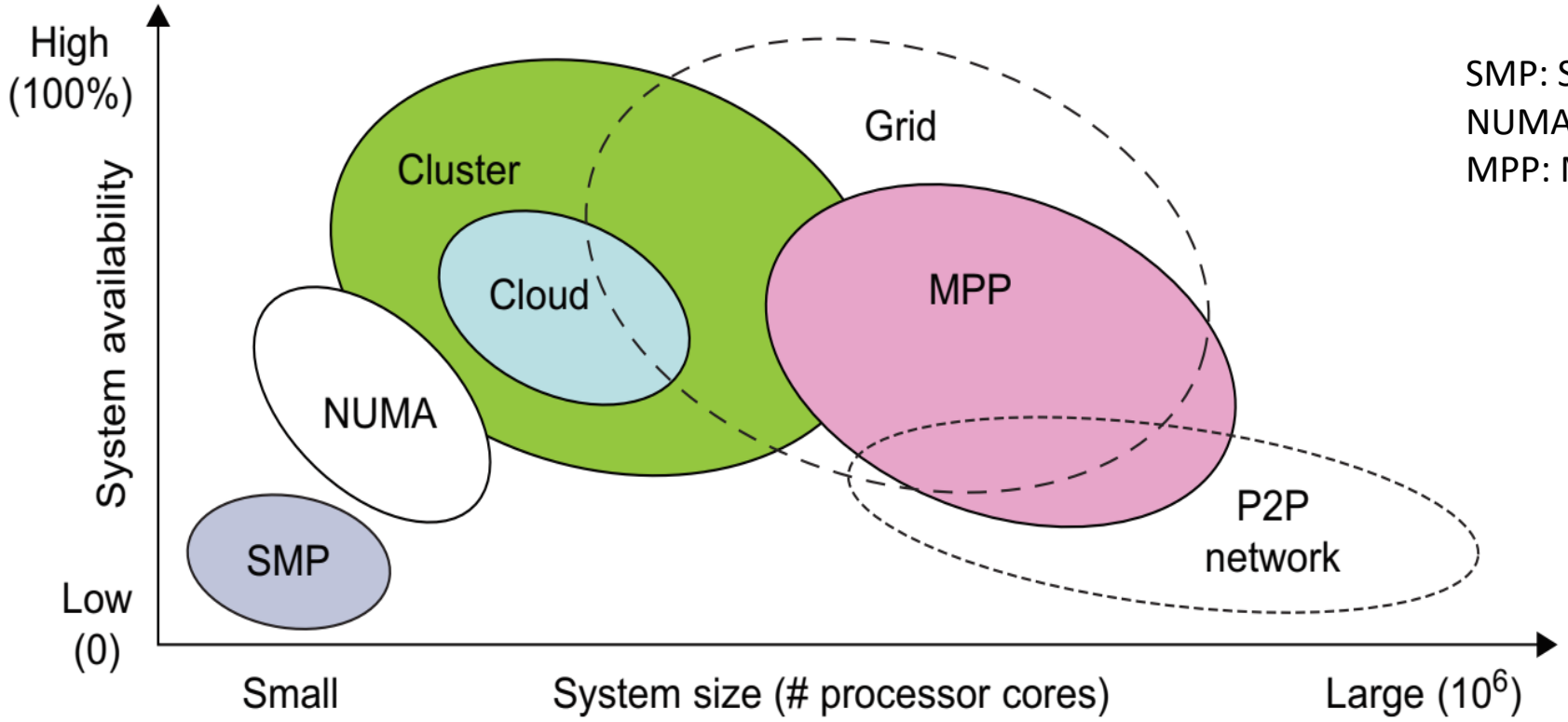
Failover is the constant capability to automatically and seamlessly switch to a highly reliable backup.
This can be operated in a redundant manner or in a standby operational mode upon the failure of a primary server, application, system or other primary system component.

System Availability

- Meanwhile, private clouds are created out of virtualized data centers; hence, a cloud has an estimated availability similar to that of the hosting cluster.
- A grid is visualized as a hierarchical cluster of clusters.
- Grids have higher availability due to the isolation of faults.
- Clusters, clouds, and grids have decreasing availability as the system increases in size.
- *A P2P file-sharing network operates independently with low availability, and even many peer nodes depart or fail simultaneously*

CLOUD COMPUTING

System Availability



SMP: Symmetric MultiProcessing
NUMA: Non Uniform Memory access
MPP: Massively Parallel Processing

Ref: T1: 1.5.2.1

Availability

- System availability

  percentage of time the system is up and running normally

$$System\ Availability = MTTF/(MTTF + MTTR)$$

- Mean Time To Failures

  expected time to failure for a non-repairable system

  MTTF= total time of correct operation(uptime) in a period/number of tracked items

- Mean Time To Repair

  average time to repair and restore a failed system

  MTTR= total hours of downtime caused by system failures/number of failures

- ***A system is highly available if it has a long mean time to failure (MTTF) and a short mean time to repair (MTTR)***

Availability

Eg -

Assume you tested 3 identical systems

starting from time 0 until all of them failed.

The first system failed at 10 hours, the second failed at 12 hours and the third

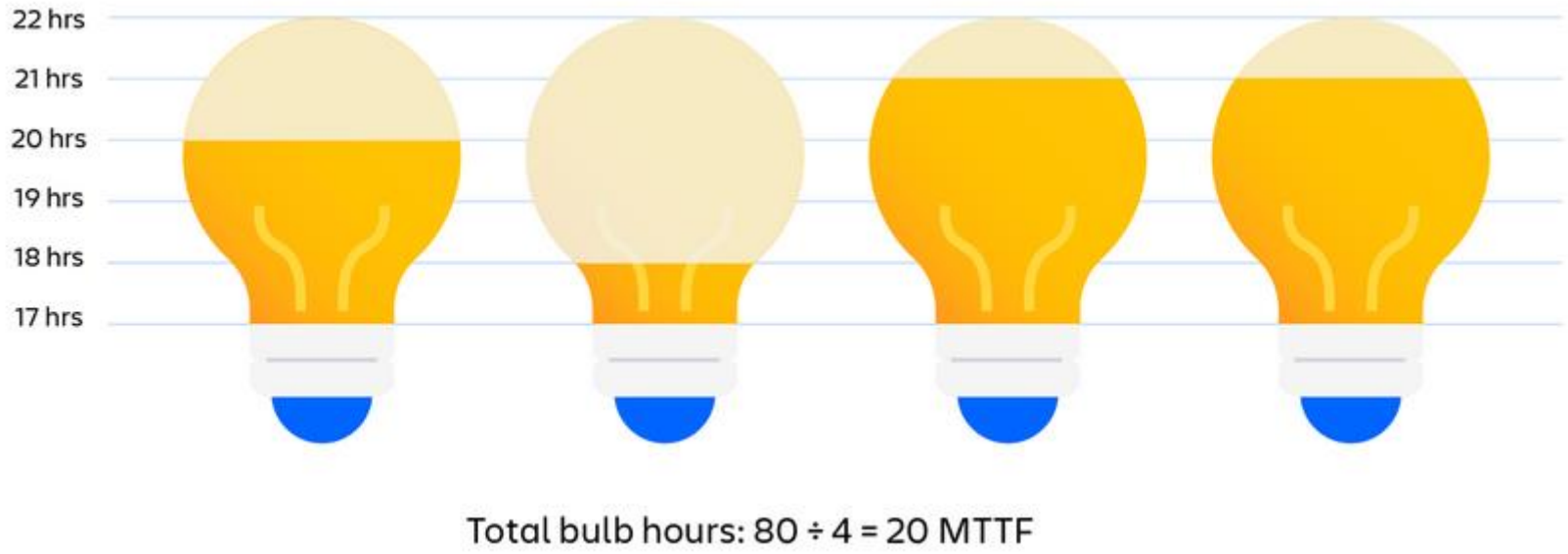
failed at 13 hours.

The MTTF is the average of the three failure times=11.6667 hours.

https://www.xplg.com/what-is-mttf-mean-time-to-failure/

Availability

For example:
Let's say you're figuring out the MTTF of light bulbs. How long do Brand Y's light bulbs last on average before they burn out?
Let's further say you have a sample of four light bulbs to test.
Light bulb A lasts 20 hours. Light bulb B lasts 18. Bulb C lasts 21. And bulb D lasts 21 hours



MTTF is 20 hours.

MTTR - Sum of all downtime in a specific period/number of incidents.

So, let's say our systems were down for 30 minutes in two separate incidents in a 24-hour period.

MTTR is 15 minutes.

Eg- a system fails three times in a month, and the failures resulted in a total of six hours of downtime, the MTTR would be two hours.

MTTR = 6 hours / 3 failures = 2 hours

While repairs can take minutes or days to complete, depending on the severity of the failure, MTTR of IT systems is typically measured in hours.

https://www.xplg.com/what-is-mttf-mean-time-to-failure/

CLOUD COMPUTING

High availability

High availability *is quality of a system or component that assures a high level of operational performance for a given period of time.*

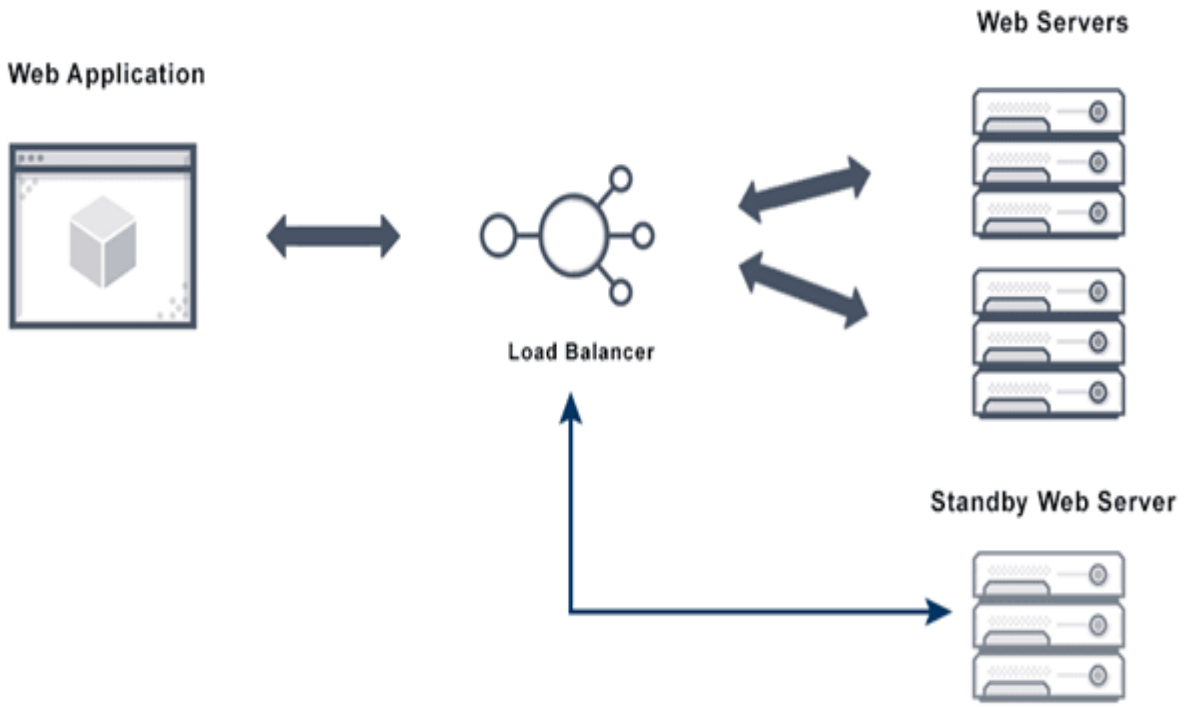
- For instance, a system that guarantees 99% of availability in a period of one year can have up to 3.65 days of downtime (1%).
- One of the goals is to eliminate single points of failure in your infrastructure. A single point of failure is a component of your technology stack that would cause a service interruption if it became unavailable.
- When setting up robust production systems, minimizing downtime and service interruptions is often a high priority. Regardless of how reliable your systems and software are, problems can occur that can bring down your applications or your servers.

CLOUD COMPUTING

What is Fault tolerance?

Fault Tolerance

- simply means a system's ability to continue operating uninterrupted despite the failure of one or more of its components.
- refers to how an operating system (OS) responds to and allows for software or hardware malfunctions and failures.



Ref - https://www.imperva.com/learn/availability/fault-tolerance/

High Availability vs Fault Tolerance

Consider the following analogy to better understand the difference between fault tolerance and high availability.

- A twin-engine airplane is a **fault tolerant system** – if one engine fails, the other one kicks in, allowing the plane to continue flying. Conversely, a car with a spare tire is **highly available**. A flat tire will cause the car to stop, but **downtime is minimal** because the tire can be easily replaced.

- Some important considerations when creating fault tolerant and high availability systems in an organizational setting include:

    1. **Downtime** – A highly available system has a minimal allowed level of service interruption. For example, a system with “five nines” availability is down for approximately 5 minutes per year. A fault-tolerant system is expected to work continuously with no acceptable service interruption.

High Availability vs Fault Tolerance

2. **Scope** – High availability builds on a shared set of resources that are used jointly to manage failures and minimize downtime. Fault tolerance relies on power supply backups, as well as hardware or software that can detect failures and instantly switch to redundant components.

2. **Cost** – A fault tolerant system can be costly, as it requires the continuous operation and maintenance of additional, redundant components. High availability typically comes as part of an overall package through a service provider (e.g., load balancer provider).

Some of your systems may require a fault-tolerant design, while high availability might suffice for others. You should weigh each system's 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.

Fault tolerance

Fault-tolerant systems use backup components that automatically take the place of failed components, ensuring no loss of service.

These include:

- ***Hardware systems***
  1. backed up by identical or equivalent systems.
  2. The interconnection network should provide some mechanism to tolerate link or switch failures.
  3. Multiple paths should be established between any two server nodes in a data center.
  4. Fault tolerance of servers is achieved by replicating data and computing among redundant servers. Similar redundancy technology should apply to the network structure

     Eg- Identical Server mirroring all the operations to the backup server.

Fault tolerance

- ***Software systems*** that are backed up by other software instances.

  Need to be aware of network failures. Packet forwarding should avoid using broken links.

  The network support software drivers should handle this transparently without affecting cloud operations.

  Power sources that are made fault tolerant using alternative sources.

  For example: many organizations have power generators

CLOUD COMPUTING

Design of Fault tolerant Systems

- Fault-tolerant systems *increase* the rate of faults by triggering them deliberately
- For example, by randomly killing individual processes without warning; by deliberately inducing faults, you ensure that the fault-tolerance machinery is continually exercised and tested, which can increase your confidence that faults will be handled correctly when they occur naturally.
- Example Approach: **The Netflix *Chaos Monkey***

Randomly terminates virtual machine instances and containers that run inside of your production environment.

Exposing engineers to failures more frequently incentivizes them to build resilient services.

Ref: T3: Chapter 8: Trouble with Distributed Sytems

CLOUD COMPUTING

Chaos Monkey

- Built upon another Netflix-made tool, Spinnaker, open-source, multi-cloud continuous delivery platform. Spinnaker can be built on top of a cloud provider and works with all major providers.
- *The providers it supports are:*
  - *Google's App Engine*
  - *Amazon Web Services*
  - *Azure*
  - *Cloud Factory*
  - *DC/OS*
  - *Google Compute Engine*
  - *Kubernetes V2 (manifest based)*
  - *Oracle*
- Once you've installed Spinnaker, you can install Chaos Monkey.

Suite of tools

- ***Chaos Kong*** *drops a whole AWS Region.*
- ***Chaos Gorilla*** *drops a whole AWS Availability Zone.*
- ***Latency Monkey*** *simulates network outages or delays.*
- ***Doctor Monkey*** *performs health checks.*
- ***Janitor Monkey*** *identifies unused resources.*
- ***Conformity Monkey*** *identifies non-conforming instances based on a set of rules.*
- ***Security Monkey*** *tests for known vulnerabilities.*
- ***10-18 Monkey*** *detects configuration and run-time problems in instances serving customers in multiple geographic regions.*

https://www.bmc.com/blogs/chaos-monkey/

Fault tolerance patterns for microservices

1. **Use Asynchronous communication (for example, message-based communication) across internal microservices :**

- It's highly advisable not to create long chains of synchronous HTTP calls across the internal microservices because that incorrect design will eventually become the main cause of bad outages.

Ref - https://medium.com/@sksonudas/fault-tolerant-patterns-for-microservice-8d0c40f4f514

Fault tolerance patterns for microservices

- On the contrary, except for the front-end communications between the client applications and the first level of microservices or fine-grained API Gateways, it's recommended to use only asynchronous (message-based) communication once past the initial request/response cycle, across the internal microservices.
- Eventual consistency and event-driven architectures will help to minimize ripple effects. These approaches enforce a higher level of microservice autonomy and therefore prevent against the problems.

Ref - https://medium.com/@sksonudas/fault-tolerant-patterns-for-microservice-8d0c40f4f514

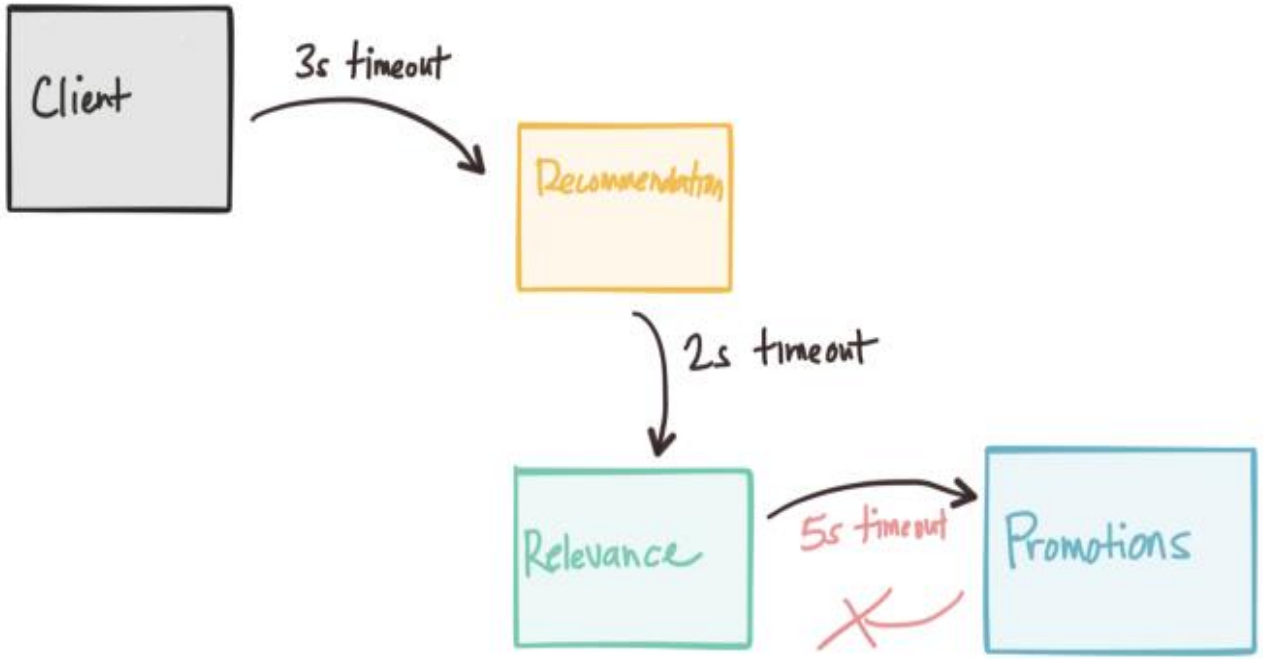
2. Network timeouts :

- Every single click in modern internet triggers multitudes of network calls and, as you probably know, network is unreliable.
- This is one of the reasons you should set request timeouts when fetching something over the wire.
- 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.

2. Network timeouts :

- Setting them is very important, because network might fail, your instance might get slow or crash.

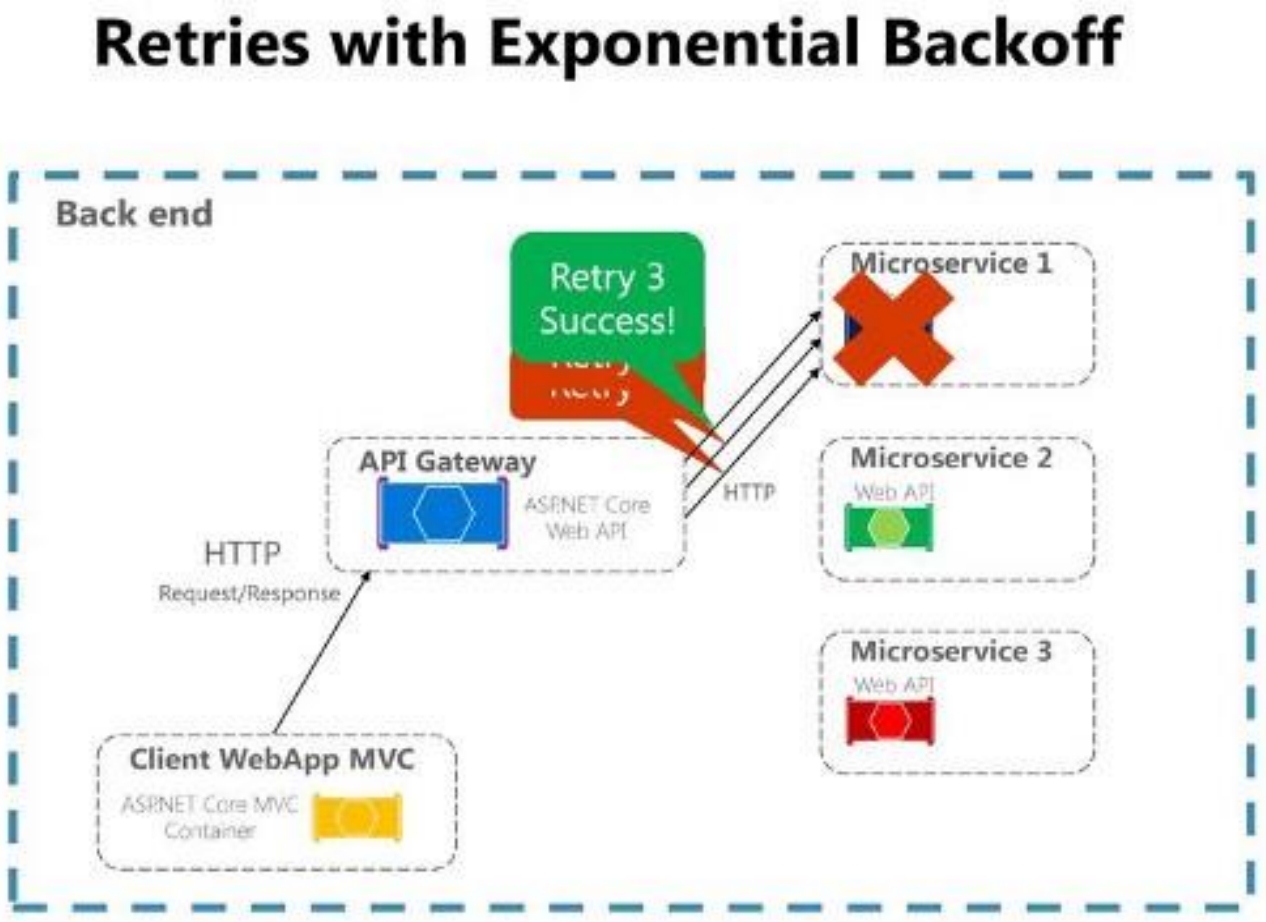
- You don't want users to wait indefinitely just because 1 of your 1000 servers suddenly shut down.



3. Use 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.
- The time between these retries is increased exponentially.

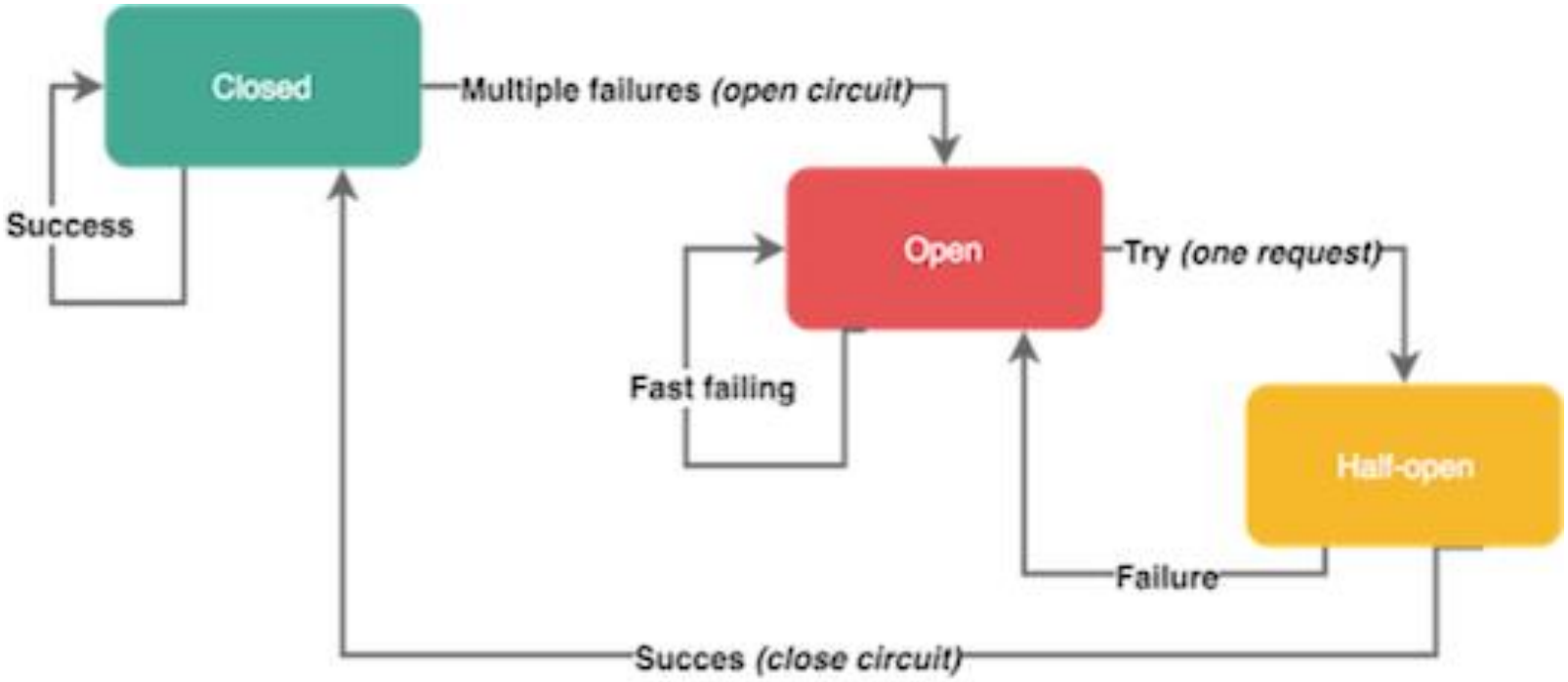
fault tolerance patterns for microservices



4. Use the Circuit Breaker pattern :

- In this approach, the client process tracks the number of failed requests.
- If the error rate exceeds a configured limit, a "circuit breaker" trips so that further attempts fail immediately.
- (If a large number of requests are failing, that suggests the service is unavailable and that sending requests is pointless.)
- After a timeout period, the client should try again and, if the new requests are successful, close the circuit breaker.

4. **Use the Circuit Breaker pattern :**

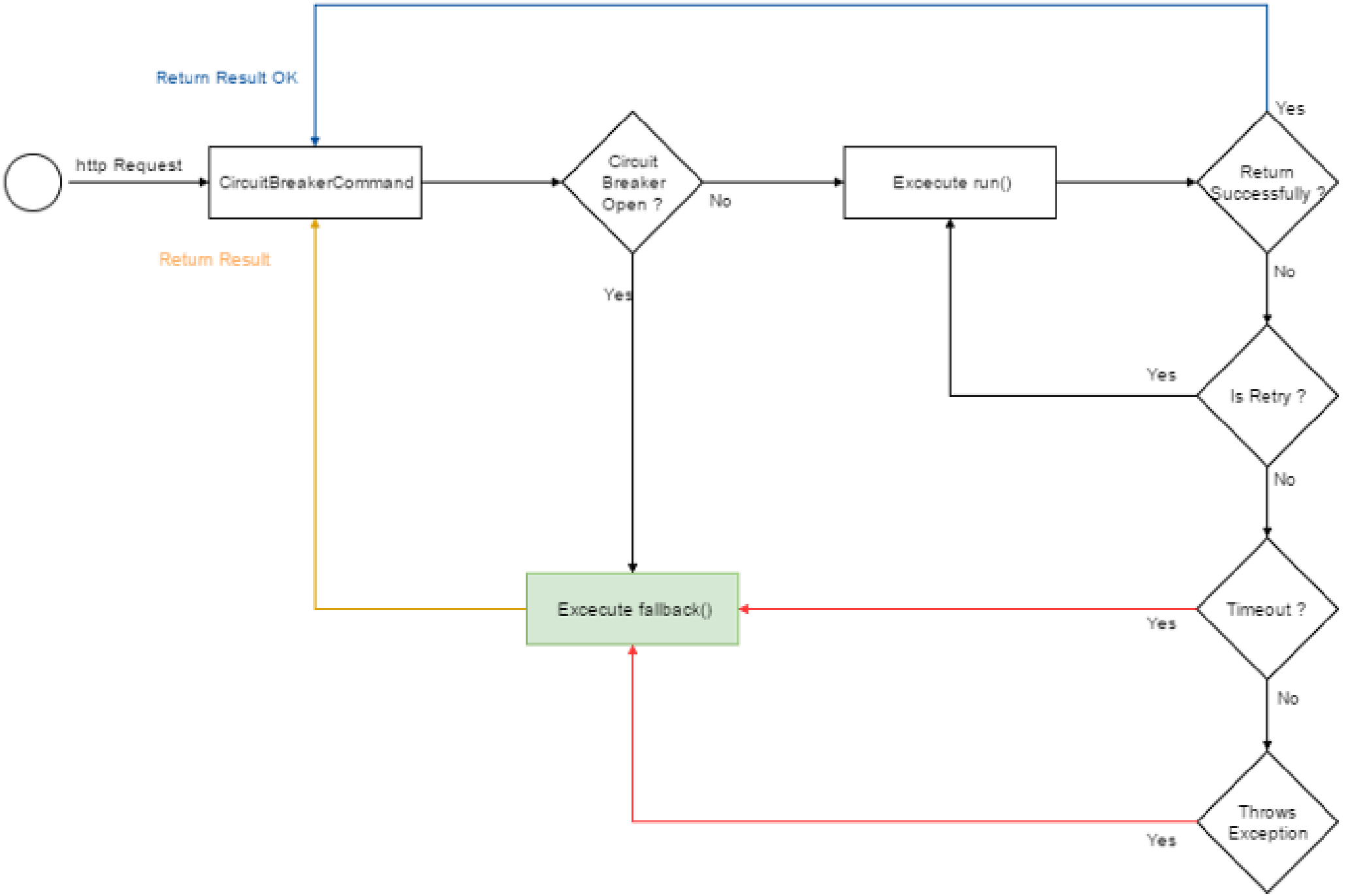


5. Provide Fallback :

- When the service request fails after retry, we need an alternative response to the request. Fallback provides an alternative solution during a service request failure.
- When the circuit breaker trips and the circuit is open, a fallback logic can be started instead.
- The fallback logic typically does little or no processing, and return value.
- Fallback logic must have little chance of failing, because it is running as a result of a failure to begin with.

5. Provide Fallback :

- 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.


Return Result OK
http Request
CircuitBreakerCommand
Circuit Breaker Open ?
No
Excecute run()
Return Successfully ?
Yes
No
Return Result
Yes
Yes
Is Retry ?
No
Excecute fallback()
Timeout ?
Yes
No
Throws Exception
Yes

6. Limit the number of queued requests :

- Clients should also 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, it's probably pointless to make additional requests, and those attempts should fail immediately.
- *Throttling or Rate limiting* is configured as a threshold on the maximum number of requests that can be made during a specific time period.
- For example, a throttle can be defined as a maximum number of requests per second, a maximum number of requests per day, or both.

Synchronous Vs Asynchronous Systems

1. A system of parallel processes is said to be synchronous if all processes run using the same clock.

2. And it is asynchronous if each process has its own independent clock.

What does Asynchronous mean?

Computation consists of steps, in each step one of the following things can happen

- A process sends a message
- A process receives a message
- A process performs some local computation.

If a process is about to perform a local computation and it is delayed then it can still do that local computation – Irrespective of what other processes do

Effect of Asynchrony

As a result it is not possible to distinguish between a slow process and a failed process.

This is the reason why consensus is not solvable in asynchronous systems

What is Consensus in Distributed Computing?

Distributed Computing

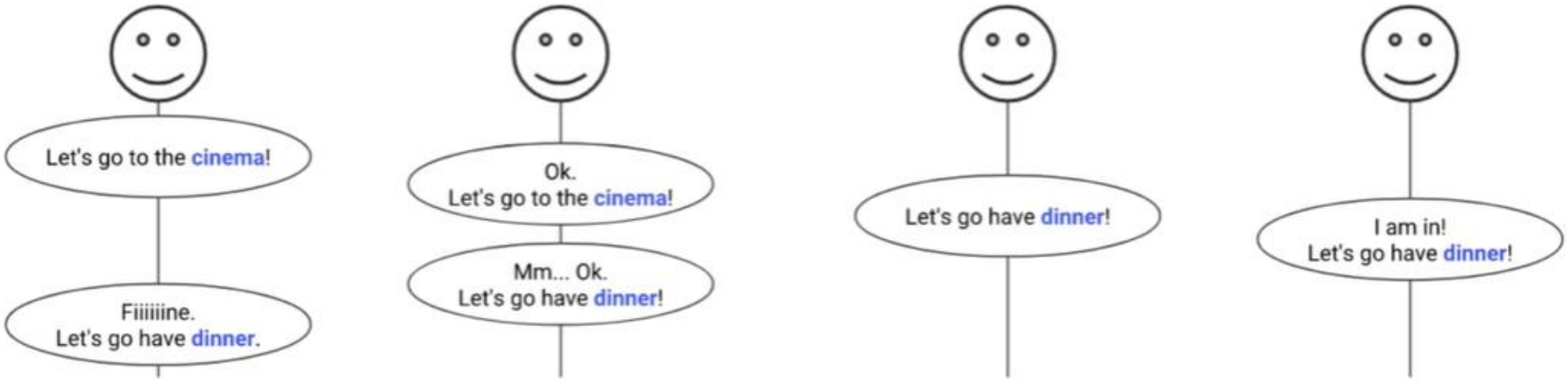
- A network of nodes operating together

Consensus

- Agreeing that something is the truth

CLOUD COMPUTING

Example



Consensus is agreeing on **one** result.
Once a **majority** agrees on a proposal, that is the consensus.
The reached consensus can be **eventually** known by everyone.
The involved parties want to agree on **any** result, not on their proposal.
Communication channels may be **faulty**, that is, messages can get lost.

What is Consensus?

Formal problem statement

- N processes
- Each process p has

  input variable xp : initially either 0 or 1

  output variable yp : initially b (can be changed only once)
- Consensus problem: design a protocol so that at the end, either:

  1. All processes set their output variables to 0 (all-0's)
  2. Or All processes set their output variables to 1 (all-1's)

What is Consensus?

- Every process contributes a value
- Goal is to have all processes decide same value

- **There might be other constraints:**

➢ **Validity** = if everyone proposes same value, then that's what's decided

➢ **Integrity** = decided value must have been proposed by some process

➢ **Non-triviality** = there is at least one initial system state that leads to each of the all-0's or all-1's outcomes

Why is it important?

Many problems in distributed systems are equivalent to consensus!

- Perfect Failure Detection
- Leader election (select exactly one leader, and every alive process knows about it)
- Agreement (harder than consensus)

So consensus is a very important problem, and solving it would be really useful!

Consensus is

- Possible to solve in synchronous systems
- Impossible to solve in asynchronous systems

How to solve consensus?

Paxos algorithm

- Most popular "consensus-solving" algorithm
- Does not solve consensus problem (which would be impossible)
- **Michael J. Fisher, Nancy A. Lynch, and Michael S. Patterson proved**
  - ***Impossibility of Distributed Consensus with One Faulty Process*** **(1985) - Dijkstra (*dike-stra*) Award (2001)**
- But provides safety and eventual liveness
- A lot of systems use it
  - Zookeeper (Yahoo!), Google Chubby, and many other companies

How Paxos works?

- A *Paxos* node can take on any or all of three roles:
  proposer, acceptor, and learner.

- A *proposer* proposes a value that it wants agreement upon.

- Each acceptor chooses a value independently — it may receive multiple proposals, each from a different *proposer* — and sends its decision to *learners*,

- Learners determine whether any value has been accepted.

- For a value to be accepted by *Paxos*, a majority of acceptors must choose the same value.

How Paxos works?

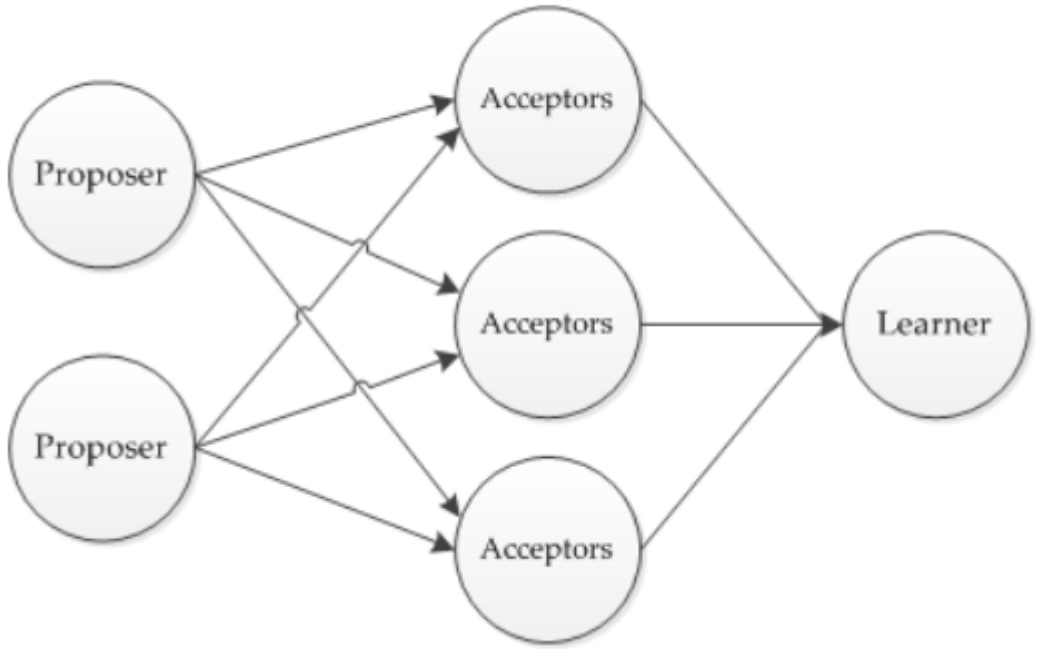


Figure 1: Basic Paxos architecture. A number of proposers make proposals to acceptors. When an acceptor accepts a value it sends the result to learner nodes.

Paxos Algorithm

- In the first 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 : Paxos Algorithm

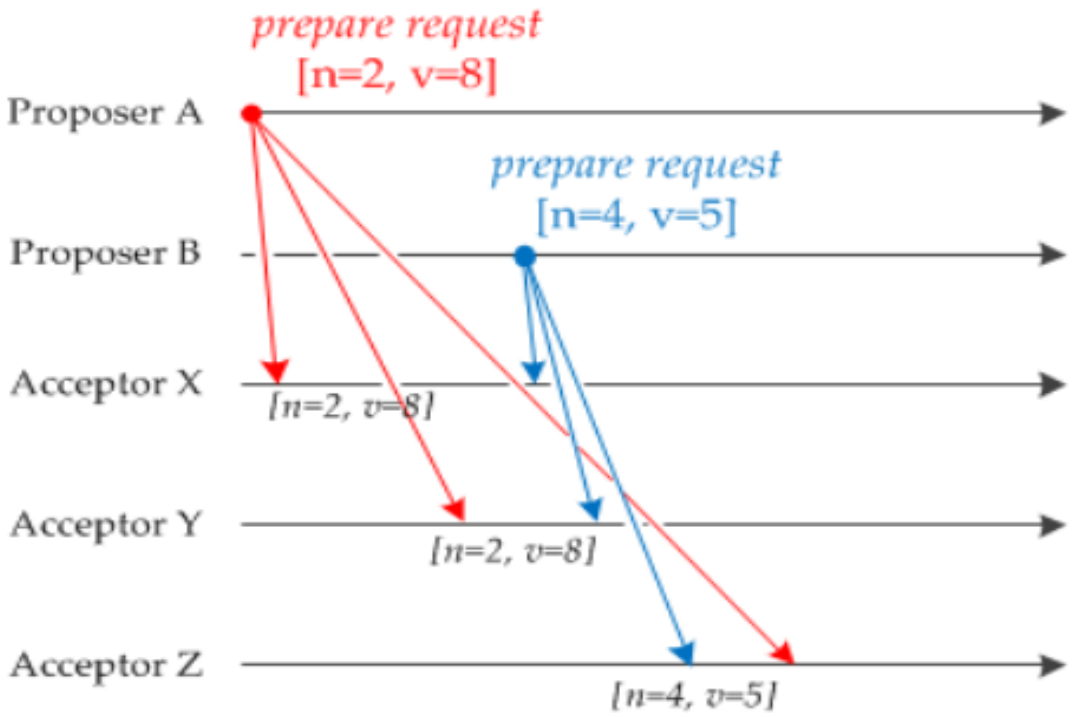


Figure 2: Proposers A and B each send prepare requests to every acceptor. In this example proposer A's request reaches acceptors X and Y first, and proposer B's request reaches acceptor Z first.

Example : Paxos Algorithm

If the acceptor receiving a *prepare request* has not seen another proposal, the acceptor responds with a *prepare response* which promises never to accept another proposal with a lower proposal number.

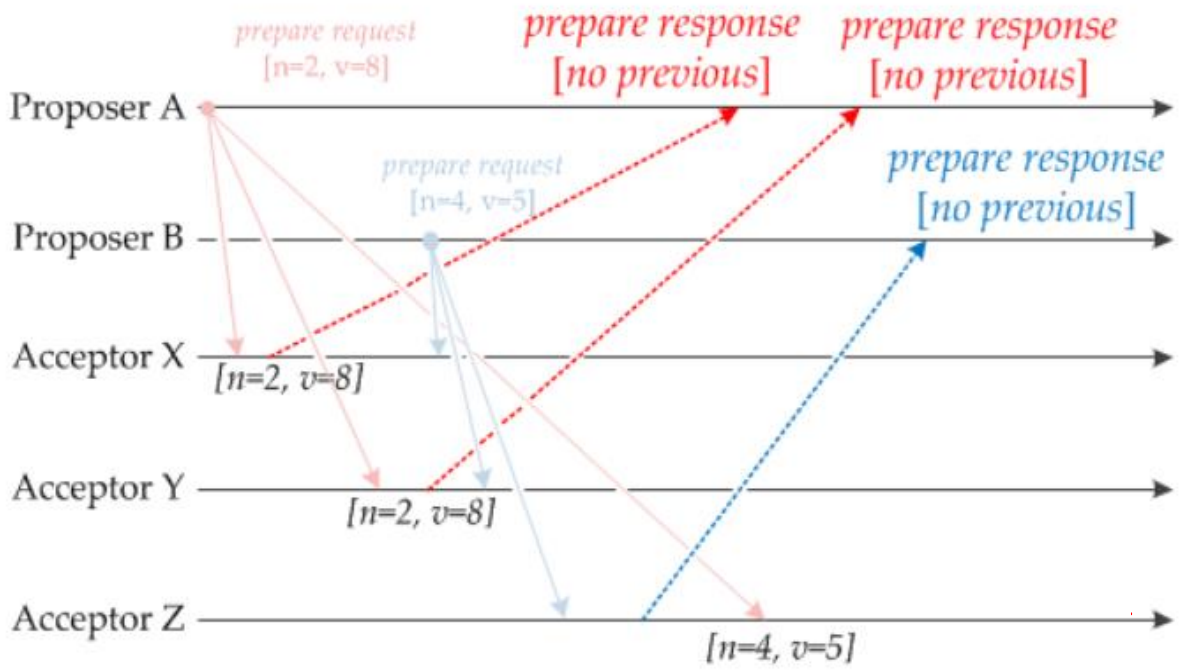


Figure 3: Paxos. Each acceptor responds to the first prepare request message that it receives.

Example : Paxos Algorithm

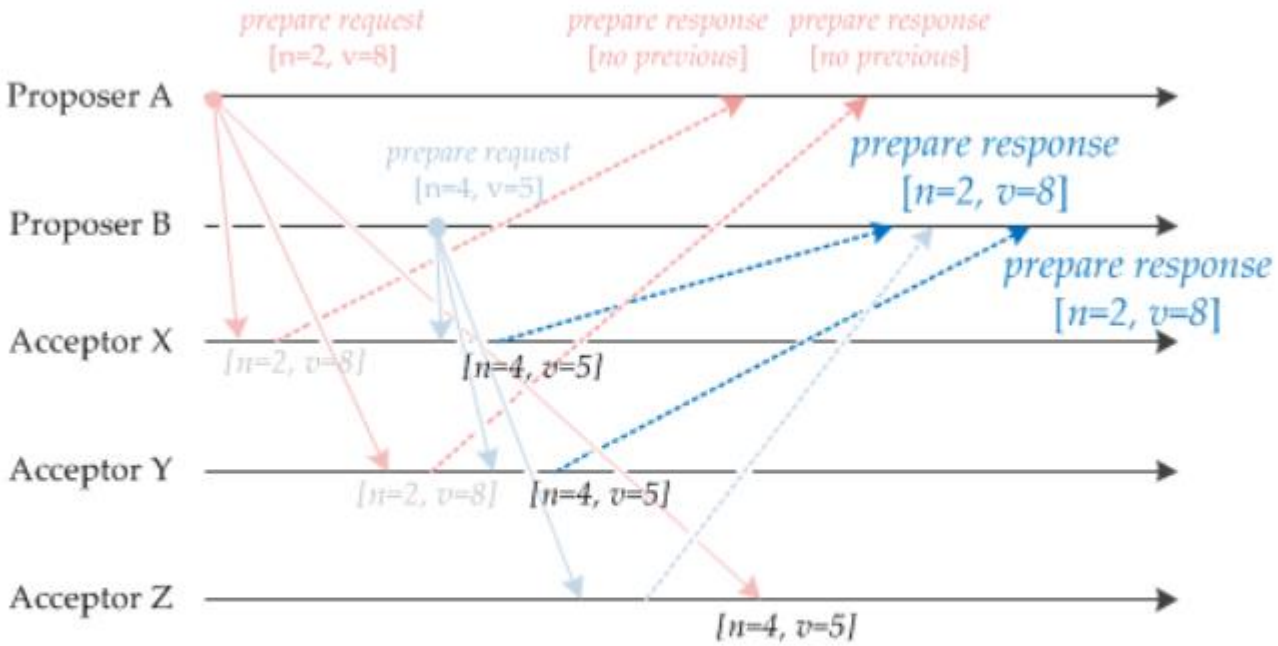


Figure 4: Acceptor Z ignores proposer A's request because it has already seen a higher numbered proposal (4 > 2). Acceptors X and Y respond to proposer B's request with the previous highest request that they acknowledged, and a promise to ignore any lower numbered proposals.

Example : Paxos Algorithm

Eventually, *acceptor Z* receives *proposer A's* request, and *acceptors X* and *Y* receive *proposer B's* request.

If the acceptor has already seen a request with a higher proposal number, the *prepare* request is ignored, as is the case with *proposer A's* request to *acceptor Z*.

If the acceptor has not seen a higher numbered request, it again promises to ignore any requests with lower proposal numbers, and sends back the highest-numbered proposal that it has accepted along with the value of that proposal.

Example : Paxos Algorithm

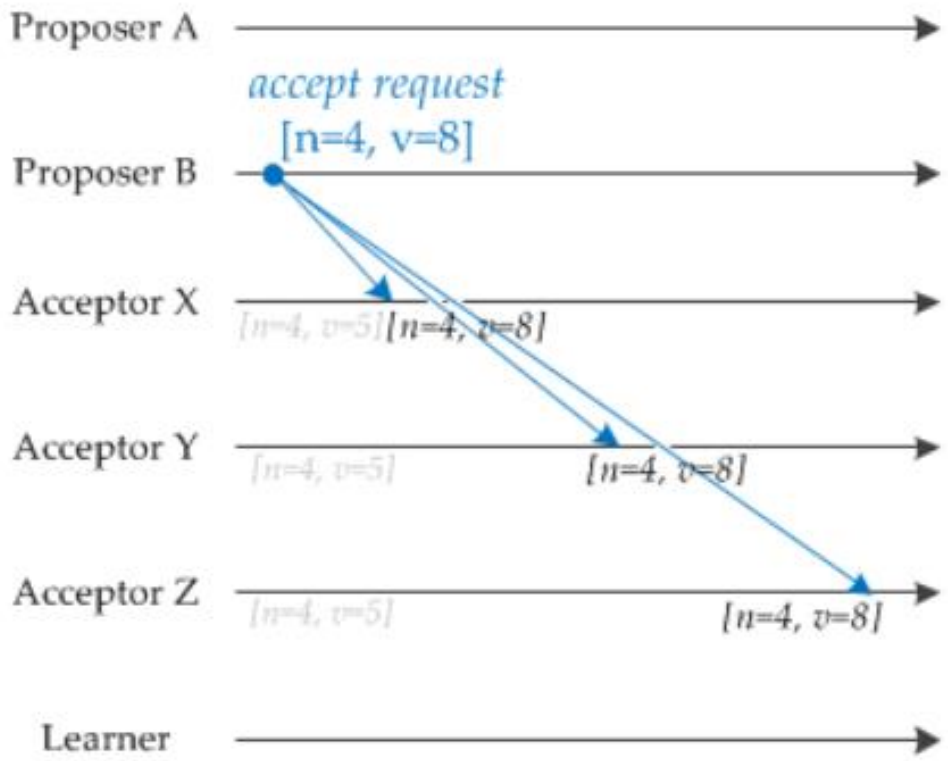


Figure 5: Proposer B sends an accept request to each acceptor, with its previous proposal number (4), and the value of the highest numbered proposal it has seen (8, from [n=2, v=8

Example : Paxos Algorithm

Once a proposer has received prepare responses from a majority of acceptors it can issue an accept request.

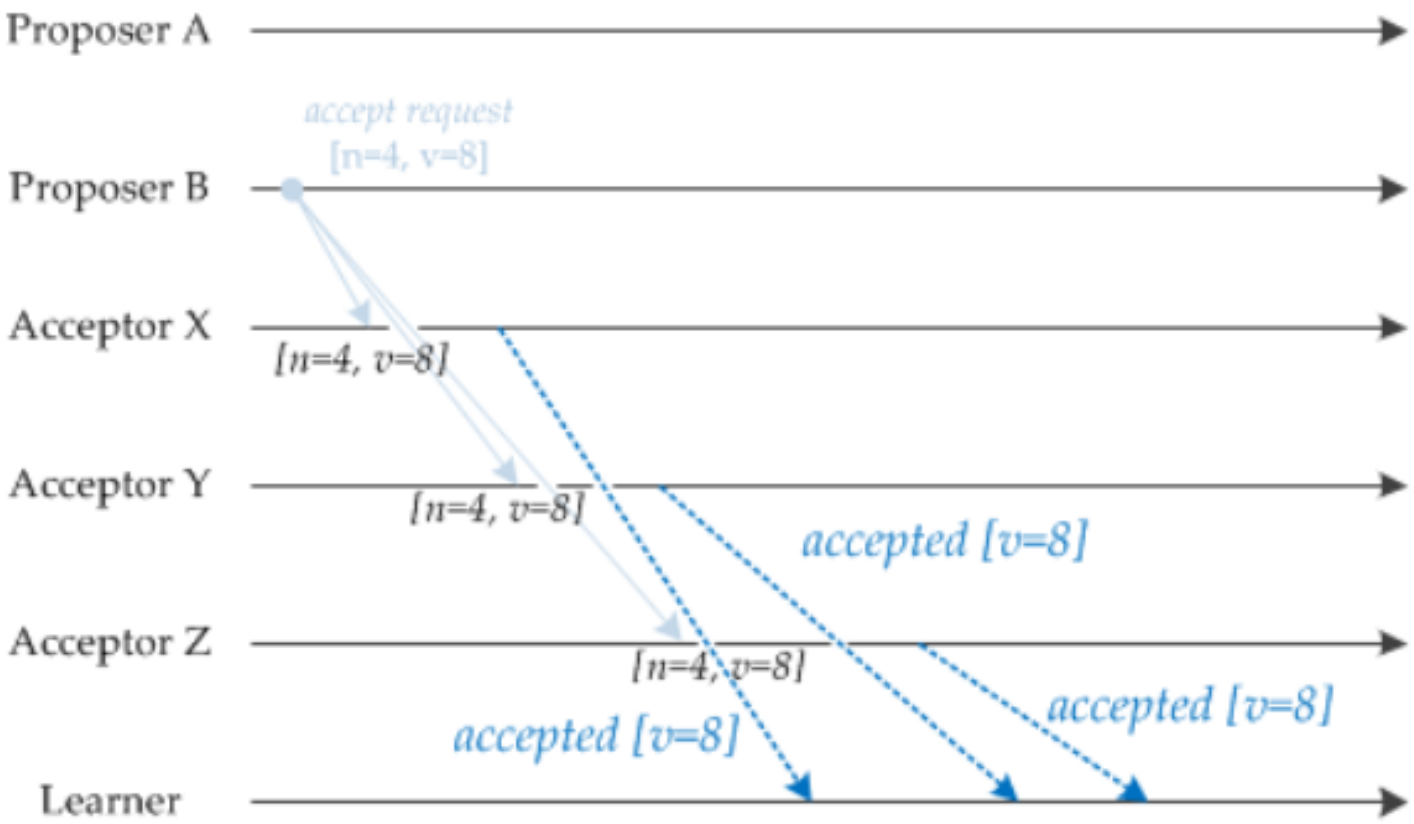
Since proposer A only received responses indicating that there were no previous proposals, it sends an accept request to every acceptor with the same proposal number and value as its initial proposal (n=2, v=8).

However, these requests are ignored by every acceptor because they have all promised not to accept requests with a proposal number lower than 4 (in response to the prepare request from proposer B).

Proposer B sends an accept request to each acceptor containing the proposal number it previously used (n=4) and the value associated with the highest proposal number among the prepare response messages it received (v=5).

Note that this is not the value that proposer B initially proposed, but the highest value from the prepare response messages it saw.

Example : Paxos Algorithm



Example : Paxos Algorithm

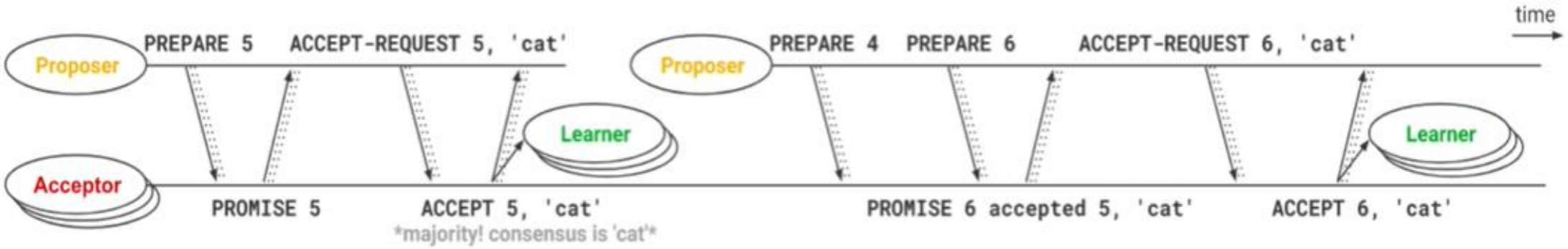
If an acceptor receives an *accept request* for a higher or equal proposal number than it has already seen, it accepts and sends a notification to every learner node.

A value is chosen by the Paxos algorithm when a learner discovers that a majority of acceptors have accepted a value.

Once a value has been chosen by Paxos, further communication with other proposers cannot change this value.

Example2 : Paxos Algorithm

The Paxos Algorithm



⇨ **Proposer** wants to propose a certain value:
It sends **PREPARE IDp** to a majority (or all) of **Acceptors**.
IDp must be unique, e.g. slotted timestamp in nanoseconds.
e.g. **Proposer** 1 chooses IDs 1, 3, 5...
Proposer 2 chooses IDs 2, 4, 6..., etc.
Timeout? retry with a new (higher) IDp.

⇨ **Acceptor** receives a **PREPARE** message for IDp:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Will promise to ignore any request lower than IDp.
Has it ever accepted anything? (assume accepted ID=**IDa**)
Yes ->Reply with **PROMISE IDp accepted IDa, value**.
No -> Reply with **PROMISE IDp**.

1 If a majority of acceptors promise, no ID<IDp can make it through.

⇨ **Proposer** gets majority of **PROMISE** messages for a specific IDp:
It sends **ACCEPT-REQUEST IDp, VALUE** to a majority (or all) of **Acceptors**.
Has it got any already accepted value from promises?
Yes - > It picks the value with the highest **IDa** that it got.
No -> It picks any value it wants.

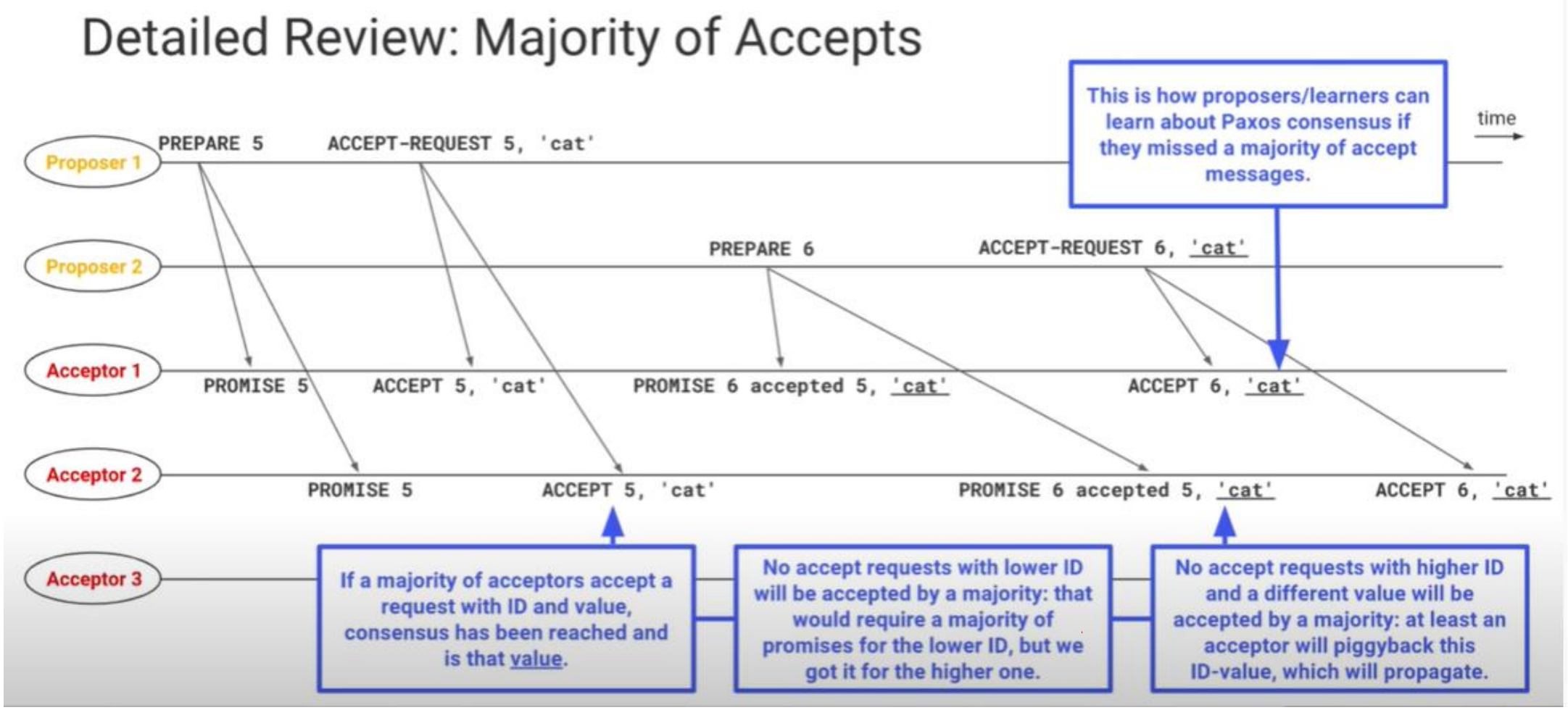
⇨ **Acceptor** receives an **ACCEPT-REQUEST** message for IDp, value:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Reply with **ACCEPT IDp, value**. Also send it to all **Learners**.

2 If a majority of acceptors accept IDp, value, consensus is reached.
Consensus is and will always be on value (not necessarily IDp).

⇨ **Proposer** or **Learner** get **ACCEPT** messages for IDp, value:

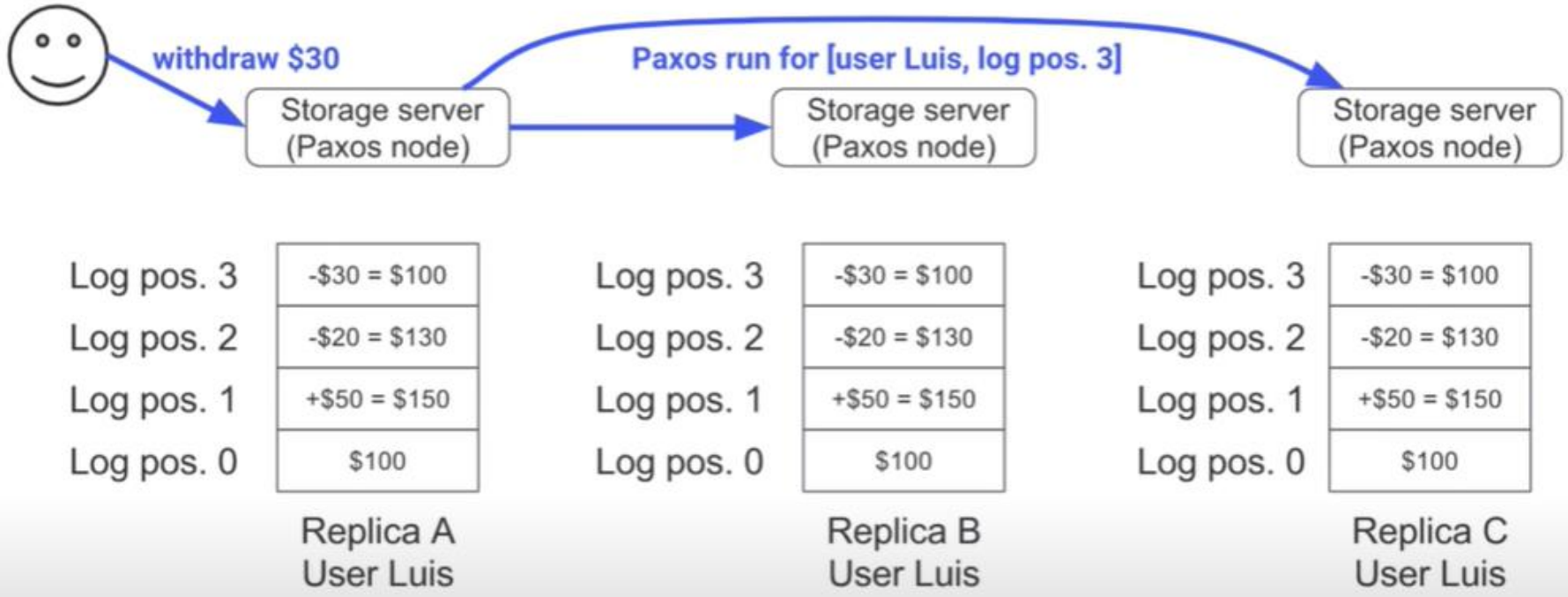
3 If a proposer/learner gets majority of accept for a specific IDp,
they know that consensus has been reached on value (not IDp).

Example2 : Paxos Algorithm



Paxos Algorithm in Practical Case:Example

Practical case: Distributed storage system based on Paxos



* The fictional storage system presented here is an extreme simplification of some parts of Megastore. Original Megastore paper:
Baker, J.; Bond, C.; Corbett, J.; Furman, J. J.; Khorlin, A.; Larson, J.; Leon, J.-M.; Li, Y.; Lloyd, A. & Yushprakh, V. (2011), Megastore: Providing Scalable, Highly Available Storage for Interactive Services., in 'CIDR' , www.cidrdb.org, , pp. 223-234 .

Summary

- **Paxos protocol:**

  It is widely used implementation of a safe, eventually-live consensus protocol for asynchronous systems

– Paxos (or variants) used in Apache Zookeeper, Google's Chubby system, Active Disk Paxos, and many other cloud computing systems

Additional References

1. L. Lamport, **"Paxos Made Simple"** in ACM SIGACT News, vol. 32, no. 4, pp. 18–25, 2001.

2. Baker, J., Bond, C., Corbett, J. C., Furman, J., Khorlin, A., Larson, J., Léon, J. M., **"Megastore: Providing Scalable, Highly Available Storage for Interactive Services"** in Proceedings of the Conference on Innovative Data Systems Research, pp. 223–234, 2011.

3. T. D. Chandra, R. Griesemer, and J. Redstone, **"Paxos made live: an engineering perspective"**, in Proceedings of the twenty-sixth annual ACM Symposium on Principles of Distributed Computing, 2007, pp. 398–407.

4.https://www.coursera.org/lecture/cloud-computing/3-3-paxos-simply-0Y9In?authMode=login.

5. https://www.youtube.com/watch?v=d7nAGI_NZPk

Why Election?

Example : Your Bank maintains multiple servers in their cloud, but for each customer, one of the servers is responsible, i.e., is the leader .

What if there are two leaders per customer?

- Inconsistency

What if servers disagree about who the leader is?

- Inconsistency

What if the leader crashes?

- Unavailability

What is Election?

In a group of processes, elect a *Leader* to undertake special tasks. And let everyone in the group know about this leader.

What happens when a leader fails (crashes)?

- Some (at least one) process detects this (how?)
- Then what?

Goals of Election algorithm

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

System Model

- Any process can call for an election.
- A process can call for at most one election at a time.
- Multiple processes can call an election simultaneously.
    - All of them together must yield a single leader only
- The result of an election should not depend on which process calls for it.
- Messages are eventually delivered.

Formally – Election problem

A run (execution) of the election algorithm must always guarantee at the end:

- Safety: For all non-faulty p: (p's elected = (q: a particular non-faulty process with the best attribute value) or Null)
- Liveness: For all election: (election terminates) & For all p: non-faulty process, p's elected is not Null

At the end of the election protocol, the non-faulty process with the best (highest) election *attribute value* is elected.

Attribute examples: leader has highest id or address. Fastest cpu. Most disk space. Most number of files, etc.

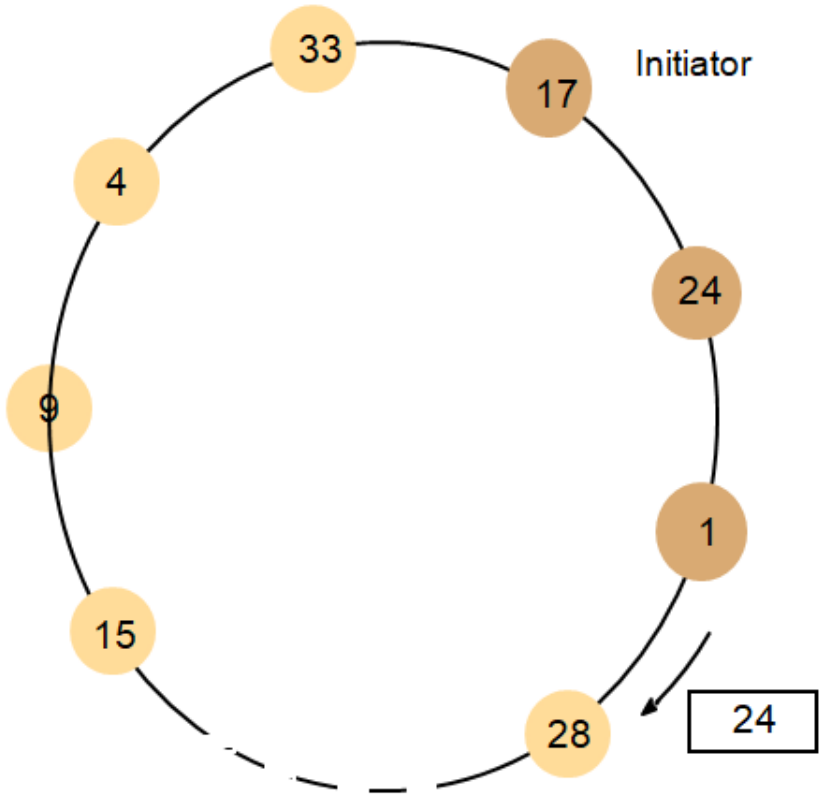
Algorithm1: Ring Election

1. N Processes are organized in a logical ring

   i) p_i has a communication channel to $p_{(i+1) \bmod N}$

   ii) All messages are sent clockwise around the ring.

2. Any process p_i that discovers the old coordinator has failed initiates an "election" message that contains p_i's own id:attr. This is the *initiator* of the election.



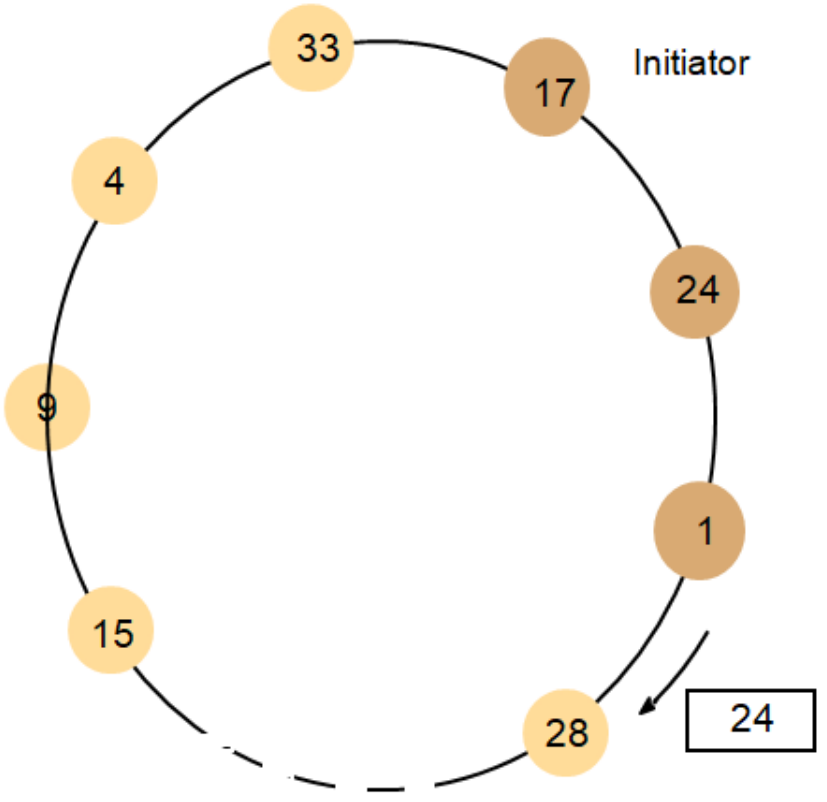
Algorithm1: Ring Election

3. When a process p_i receives an *election* message, it compares the attr in the message with its own attr.

i) If the arrived attr is greater, p_i forwards the message.

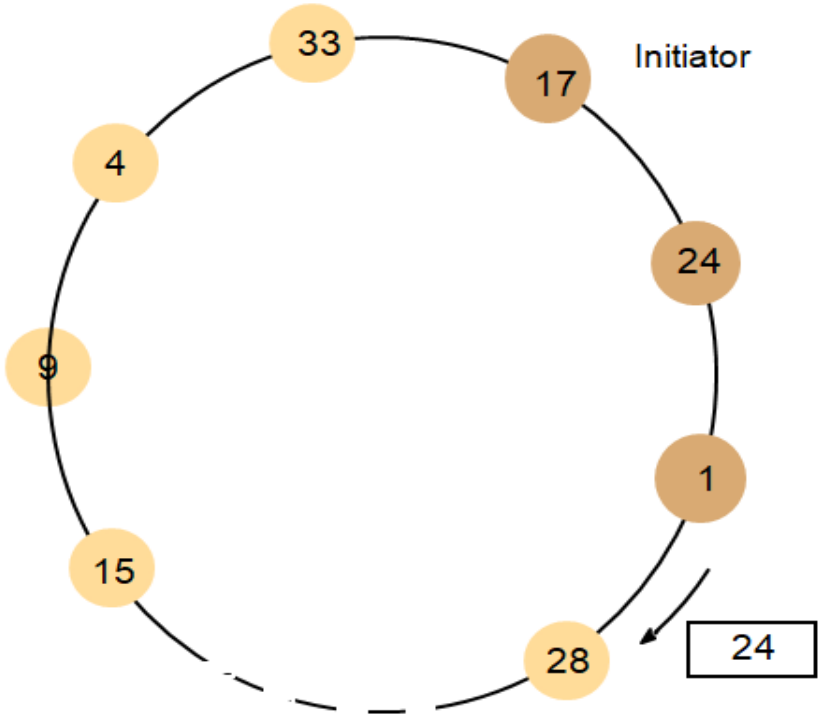
ii) If the arrived attr is smaller and p_i has not yet forwarded an election message, it overwrites the message with its own id:attr, and forwards it.

iii) If the arrived id:attr matches that of p_i, then p_i's attr must be the greatest (why?), and it becomes the new coordinator. This process then sends an "elected" message to its neighbor with its id, announcing the election result.



Algorithm1: Ring Election

4. When a process p_i receives an elected message, it
   i) sets its variable $elected_i$ ← id of the message.
   ii) forwards the message, unless it is the new coordinator.

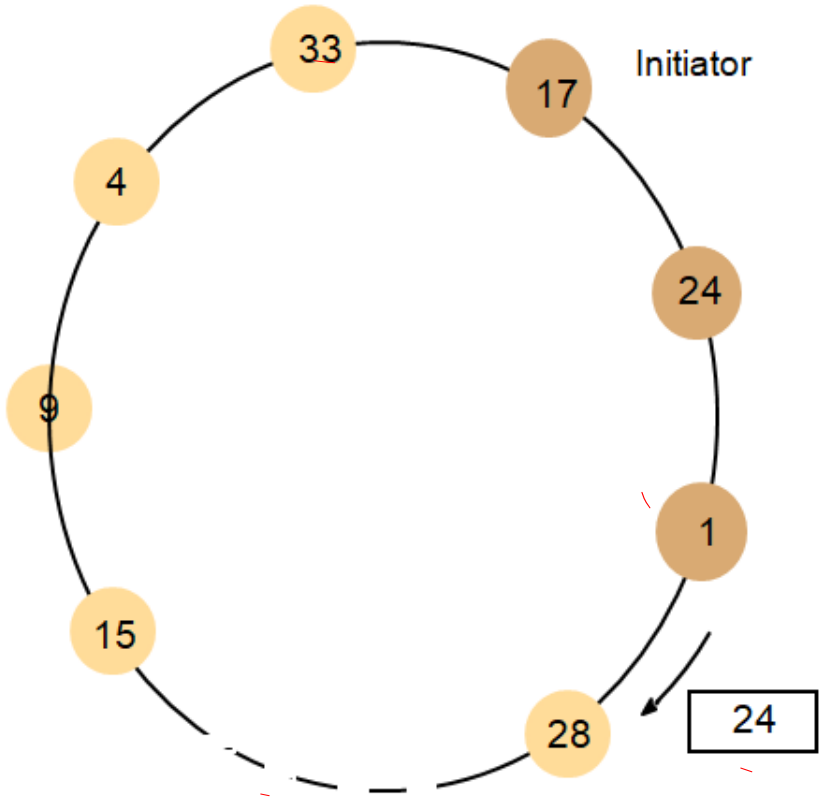


Example:

(In this example, attr:=id)

- In the example: The election was started by process 17.
  The highest process identifier encountered so far is 24.
  (final leader will be 33)

- The worst-case scenario occurs when the counter-clockwise neighbor (@ the initiator) has the highest attr.

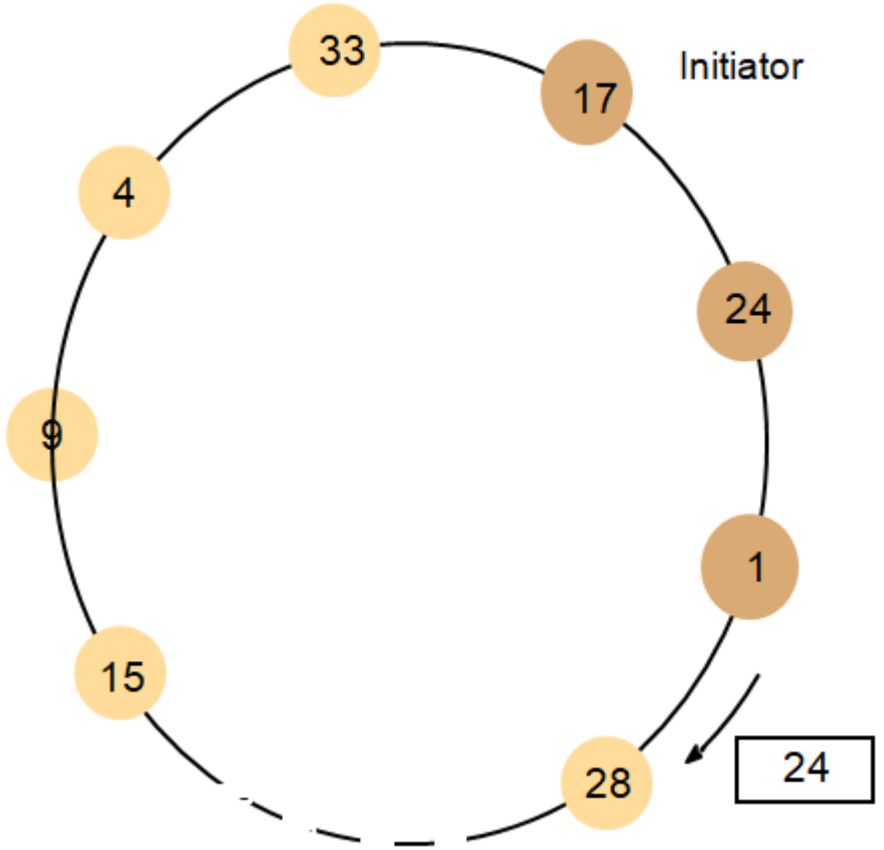


Example –(Cont…)

- **The worst-case scenario occurs when the counter-clockwise neighbor has the highest attr.**

In a ring of N processes, in the worst case:

- A total of N-1 messages are required to reach the new coordinator-to-be(election messages).
- Another N messages are required until the new coordinator-to-be ensures it is the new coordinator (election messages – no changes).
- Another N messages are required to circulate the elected messages.
- Total Message Complexity = 3N-1
- Turnaround time = 3N-1

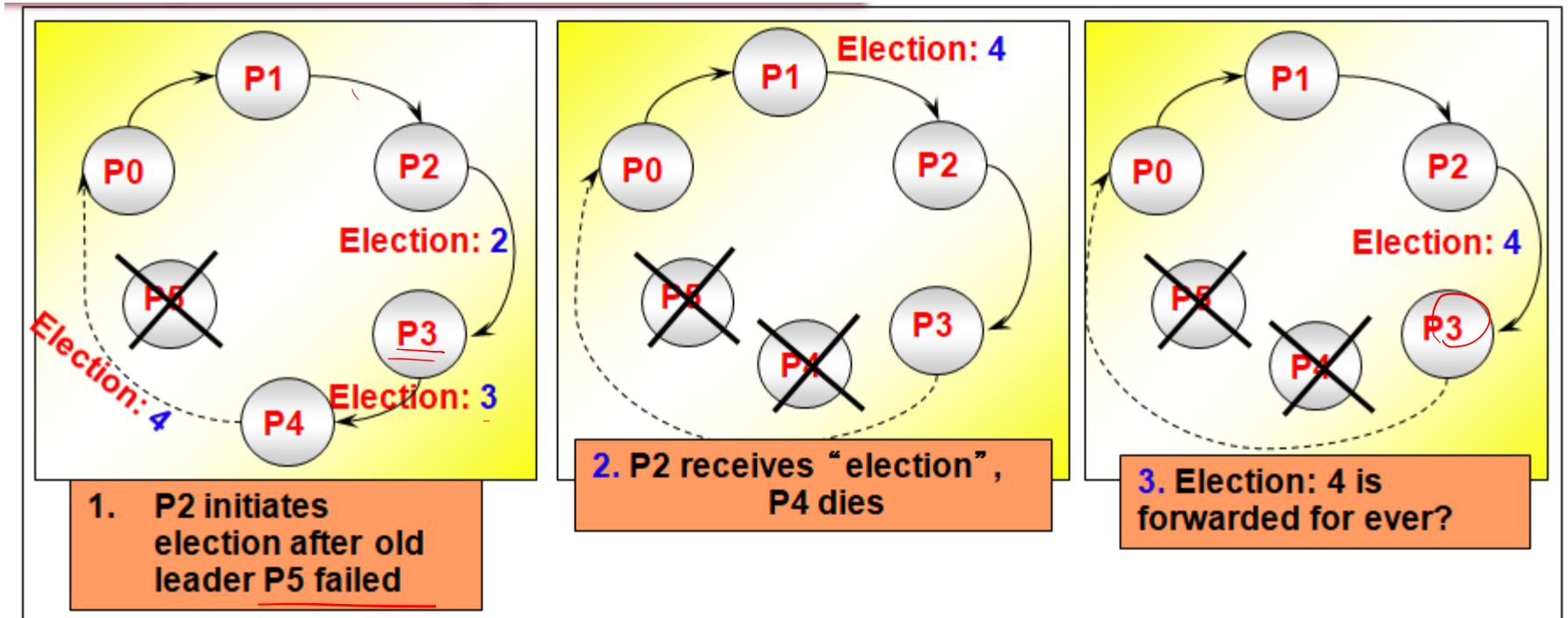


Correctness?

In the above example – no failures happen during the run of the election algorithm. Safety and Liveness are satisfied.

What happens if there are failures during the election run?

Correctness?



May not terminate when process failure occurs during the election!
Consider above example where attr == id

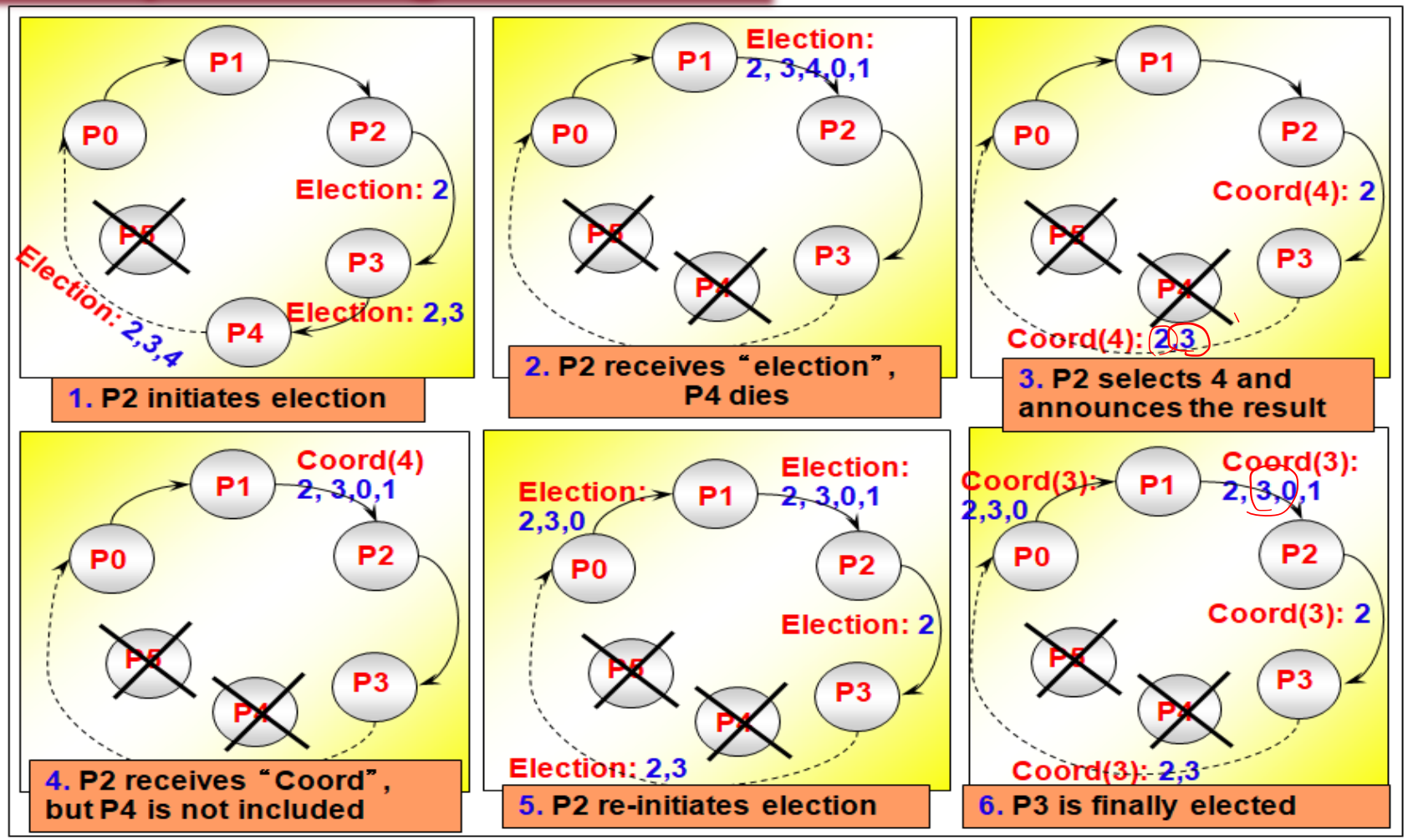
Does not satisfy liveness

Algorithm 2: Modified Ring Election

- Processes are organized in a logical ring.
- Any process that discovers the coordinator (leader) has failed initiates an "election" message.
- The message is circulated around the ring, bypassing failed processes.
- Each process appends (adds) its id:attr to the message as it passes it to the next process (without overwriting what is already in the message)
- Once the message gets back to the initiator, it elects the process with the best election attribute value.
- It then sends a "coordinator" message with the id of the newly-elected coordinator. Again, each process adds its id to the end of the message, and records the coordinator id locally.
- Once "coordinator" message gets back to initiator,
  - election is over if would-be-coordinator's id is in id-list.
  - else the algorithm is repeated (handles election failure).

Algorithm 2: Modified Ring Election

Example: Ring Election



Modified Ring Election

- Supports concurrent elections – an initiator with a lower id blocks other initiators' election messages
- Reconfiguration of ring upon failures
  - Can be done if all processes "know" about all other processes in the system
- If initiator non-faulty …
  - How many messages? 2N
  - What is the turnaround time? 2N
  - Size of messages? O(N)

Leader Election is Hard

- The Election problem is related to the ***consensus problem***
- Consensus is impossible to solve with 100% guarantee in an asynchronous system with no bounds on message delays and arbitrarily slow processes
- So is leader election in fully asynchronous system model ?
- Where does the modified Ring election start to give problems with the above asynchronous system assumptions?
    - *pi* may just be very slow, but not faulty (yet it is not elected as leader!)
    - Also slow initiator, ring reorganization

Bully Algorithm

- In the bully algorithm, all the processes know the other processes ids.
- when a process finds that the coordinator or the leader has failed it can find this via the failure detector.
- If the process knows that it is the process with the next highest id after the leader, it elects itself as the new leader.
- And it sends a coordinator message to all the processes that have lower identifiers than itself.
- At this point, the election is completed. This is why this is called a bully algorithm.

Algorithm 3: Bully Algorithm

- When a process finds the coordinator has failed, if it knows its id is the highest, it elects itself as coordinator, then sends a *coordinator* message to all processes with lower identifiers than itself.

- A process initiates election by sending an *election* message to only processes that have a higher id than itself.

  - If no answer within timeout, send coordinator message to lower id processes → Done.

  - if any answer received, then there is some non-faulty higher process → so, wait for coordinator message. If none received after another timeout, start a new election.

Algorithm 3: Bully Algorithm (Cont..)

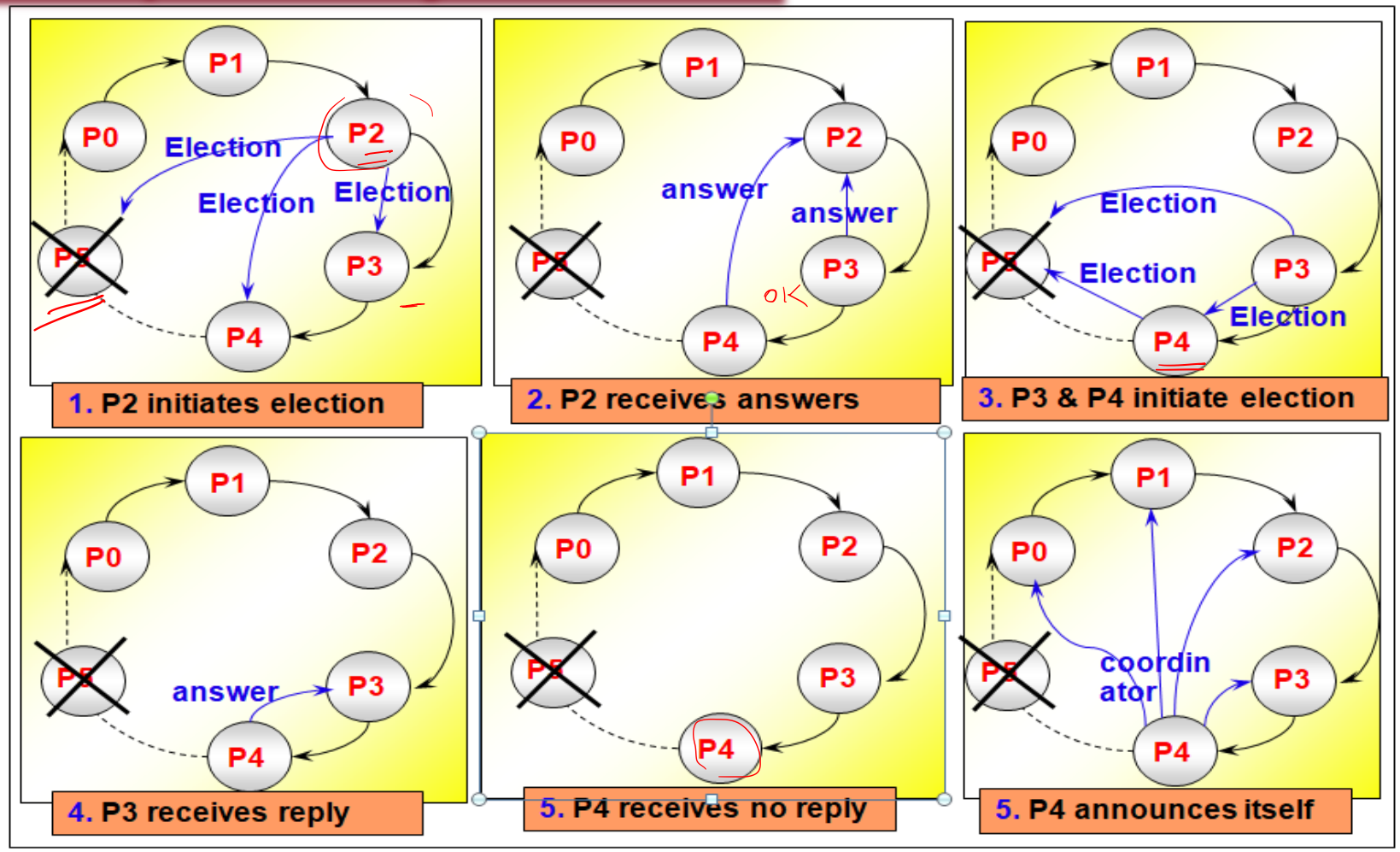
- A process that receives an "election" message replies with *answer* message, & starts its own election protocol (unless it has already done so)

Assumptions:

- **Synchronous system**
  - All messages arrive within T_{trans} units of time.
  - A reply is dispatched within $T_{process}$ units of time after the receipt of a message.
  - if no response is received in $2T_{trans} + T_{process}$, the process is assumed to be faulty (crashed).
- attr==id
- Each process knows all the other processes in the system (and thus their id's)

Algorithm 3: Bully Algorithm (Cont..)

Example: Bully Election



Steps in Bully Algorithm:

1. We start with 5 processes, which are connected to each other. Process 5 is the leader, as it has the highest number.

2. Process 5 fails.

3. Process 2 notices that Process 5 does not respond. So it starts an election, notifying those processes with ids greater than 2.

4. Then Process 3 and Process 4 respond, telling Process 2 that they'll take over from here.

5. Process 3 sends election messages to the Process 4 and Process 5.

6. Only Process 4 answers to process 3 and takes over the election.

7. Process 4 sends out only one election message to Process 5.

8. When Process 5 does not respond Process 4 ,then it declares itself the winner.

Failures During Election Run:

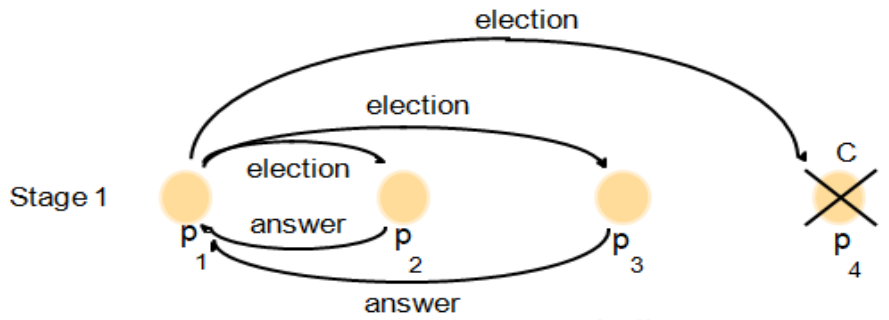
- So you can also have failures that happen during the election run.

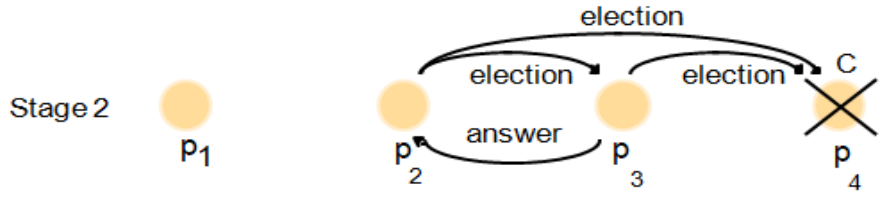
- You can also have failures that happen during the election run, so P4 might fail after it has sent the coordinator, after it has sent the election message, but before it sends the coordinator message.

Failures During Election Run:

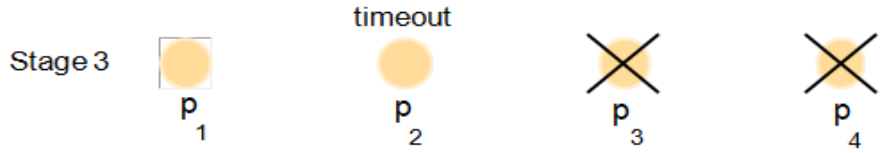
The Bully Algorithm with Failures

The coordinator p_4 fails and p_1 detects this

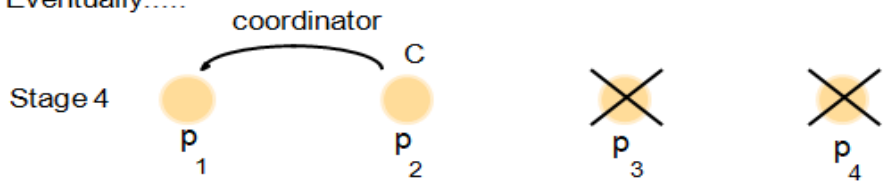




p_3 fails



Eventually.....



Analysis of Bully Algorithm

Worst case scenario: When the process with the lowest id in the system detects the failure.

- N-1 processes altogether begin elections, each sending messages to processes with higher ids.

  i-th highest id process sends i-1 election messages

- The message overhead is $O(N^2)$.
- Turnaround time is approximately 5 message transmission times if there are no failures during the run:

  1. Election message from lowest id process
  2. Answer to lowest id process from 2nd highest id process
  3. Election from 2nd highest id process
  4. Timeout for answers @ 2nd highest id process
  5. Coordinator message from 2nd highest id process

Analysis of Bully Algorithm

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.

Summary

- Coordination in distributed systems requires a leader process
- Leader process might fail
- Need to (re-) elect leader process
- Three Algorithms
  - Ring algorithm
  - Modified Ring algorithm
  - Bully Algorithm

Additional References

1. https://onlinecourses.nptel.ac.in/noc21_cs15/unit?unit=31&lesson=32

2. https://www.coursera.org/learn/cloud-computing-2/lecture/K8QwJ/1-4-bully-algorithm

3. https://www.cs.colostate.edu/~cs551/CourseNotes/Synchronization/BullyExample.html

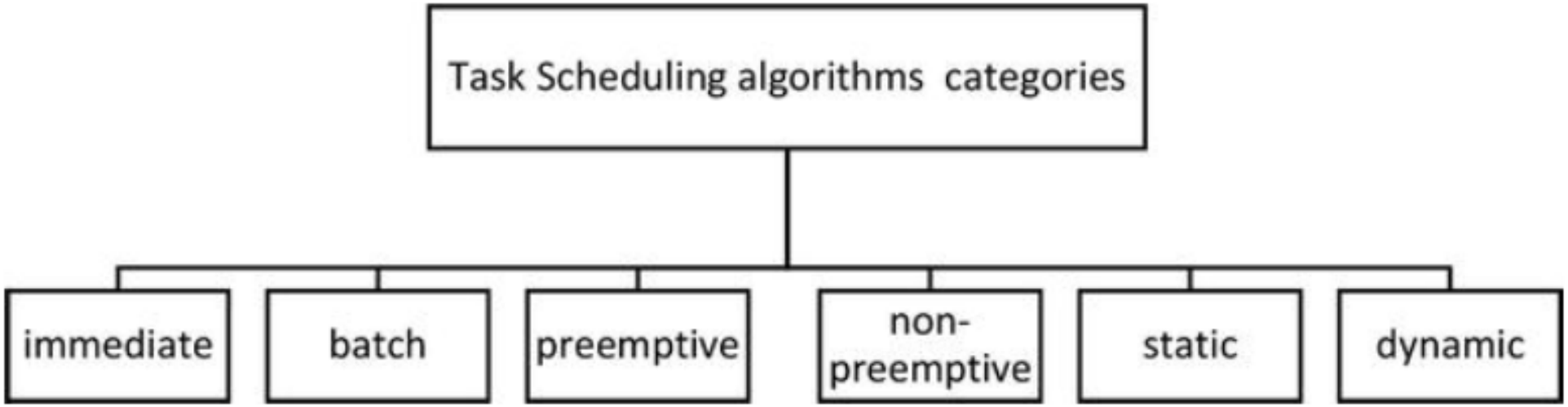
Task Scheduling Algorithms

Tasks scheduling algorithms are defined as a set of rules and policies used to assign tasks to the suitable resources (CPU, memory, and bandwidth) to get the highest level possible of performance and resources utilization.

Advantages

- Manage cloud computing performance and QoS.
- Manage the memory and CPU.
- The good scheduling algorithms maximizing resources utilization while minimizing the total task execution time.
- Improving fairness for all tasks.
- Increasing the number of successfully completed tasks.
- Scheduling tasks on a real-time system.
- Achieving a high system throughput.
- Improving load balance.

Task Scheduling Algorithms - Classification



Task Scheduling Categories

Immediate scheduling: When new tasks arrive, they are scheduled to VMs directly.

Batch scheduling: Tasks are grouped into a batch before being sent; this type is also called mapping events.

Static scheduling: It is considered very simple compared to dynamic scheduling; it is based on prior information of the global state of the system. It does not take into account the current state of VMs and then divides all traffic equivalently among all VMs in a similar manner such as round robin (RR) and random scheduling algorithms.

Task Scheduling Categories

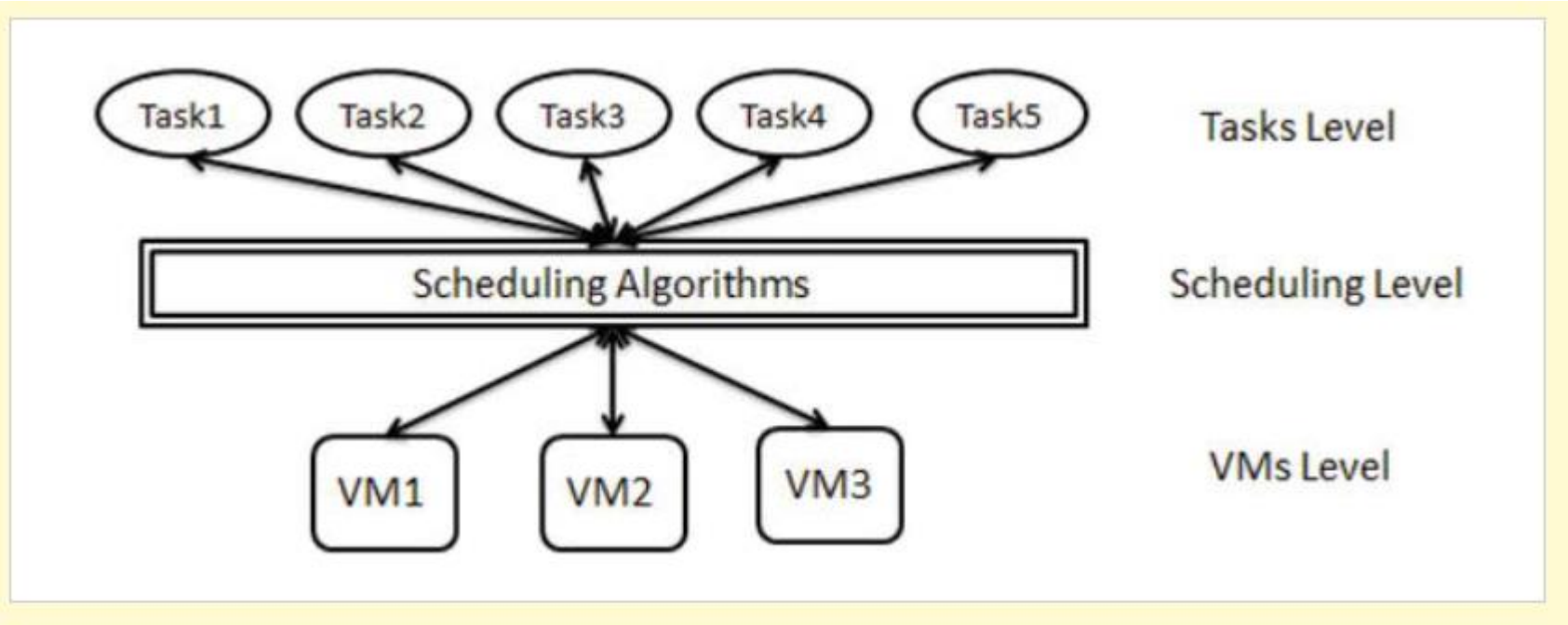
Dynamic scheduling: takes into account the current state of VMs and does not require prior information of the global state of the system and distribute the tasks according to the capacity of all available VMs.

Preemptive scheduling: each task is interrupted during execution and can be moved to another resource to complete execution.

Non-preemptive scheduling: VMs are not re-allocated to new tasks until finishing execution of the scheduled task.

CLOUD COMPUTING

Task scheduling system in cloud computing



Task scheduling system in cloud computing

It passes through three levels.

The first task level: It is a set of tasks (Cloudlets) that is sent by cloud users, which are required for execution.

The second scheduling level: It is responsible for mapping tasks to suitable resources to get highest resource utilization with minimum makespan. The makespan is the overall completion time for all tasks from the beginning to the end.

The third VMs level: It is a set of (VMs) which are used to execute the tasks

Static Tasks Scheduling Algorithms

1. FCFS
2. SJF
3. MAX-MIN

FCFS

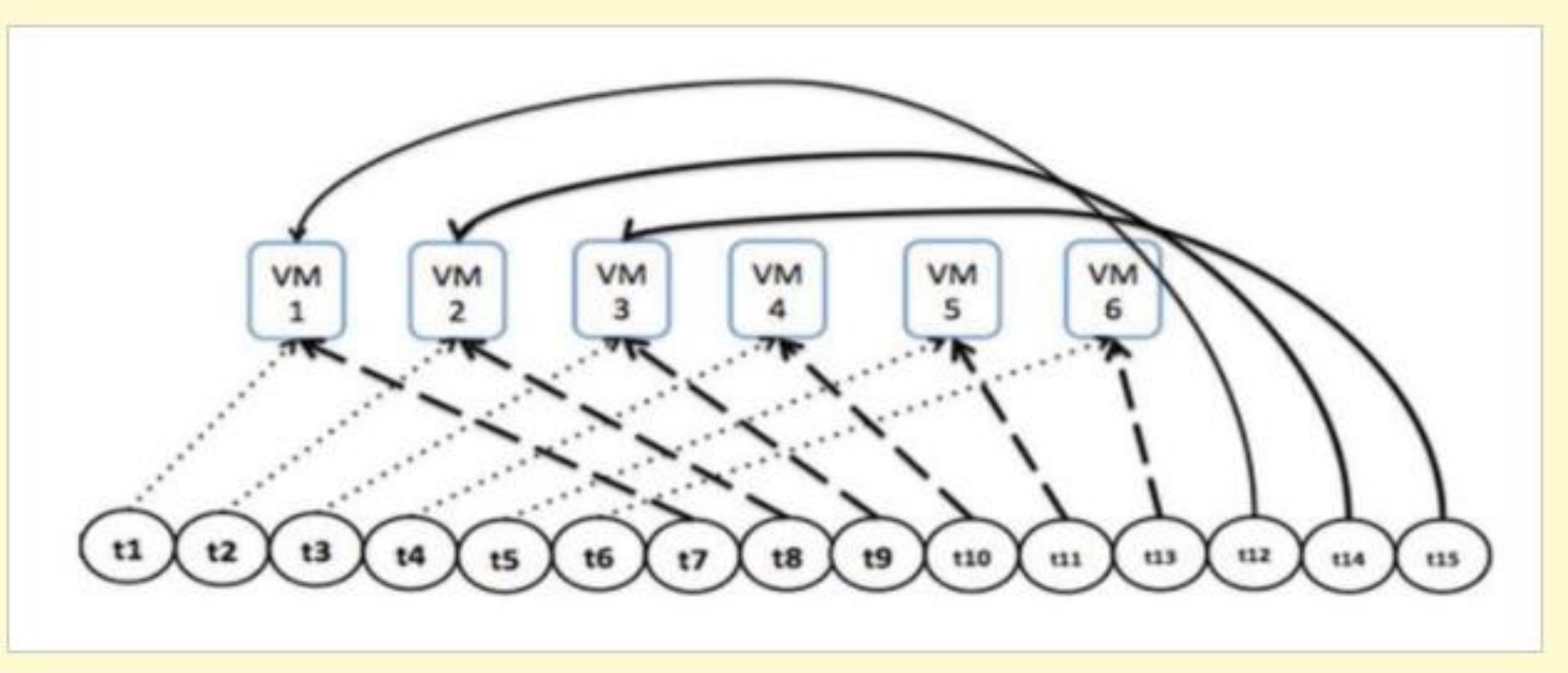
It is the order of tasks in task list is based on their arriving time then assigned to VMs.

Example:

Assume we have 15 tasks with their lengths and six VMs with different properties based on tasks size

Task	Length
t1	100000
t2	70000
t3	5000
t4	1000
t5	3000
t6	10000
t7	90000
t8	100000
t9	15000
t10	1000
t11	2000
t12	4000
t13	20000
t14	25000
t15	80000

FCFS Work Mechanism



FCFS Work Mechanism

Task	ET	Waiting time		
t1	200	VM1		
t2	140	VM2		
t3	3.33	VM3		
t4	0.66	VM4		
t5	1.2	VM5		
t6	4	VM6		
t7	180	Wait(200)	VM1	
t8	200	Wait(140)	VM2	
t9	10	Wait(3.33)	VM3	
t10	0.66	Wait(0.66)	VM4	
t11	0.8	Wait(1.2)	VM5	
t12	1.6	Wait(4)	VM6	
t13	40	Wait(380)		VM1
t14	50	Wait(340)		VM2
t15	53.33	Wait(13.33)		VM3

FCFS Work Mechanism

Advantages

Most popular and simplest scheduling algorithm.
Fairer than other simple scheduling algorithms.
Depend on FIFO rule in scheduling task.
Less complexity than other scheduling algorithms.

Disadvantages

Tasks have high waiting time.
Not give any priority to tasks.
Resources are not consumed in an optimal manner.

SJF

- Tasks are sorted based on their priority.
- Priority is given to tasks based on tasks lengths and begins from (smallest task ≡ highest priority).

SJF Work Mechanism

Example:

Assume we have 15 tasks with their lengths and six VMs with different properties based on tasks size

Tasks	t4	t10	t11	t5	t12	t3	t6	t9	t13	t14	t2	t15	t7	t1	t8
lengths	1000	1000	2000	3000	4000	5000	10000	15000	20000	25000	70000	80000	90000	100000	100000

CLOUD COMPUTING

SJF Work Mechanism

Task	ET	Waiting time		
t4	2	VM1		
t10	2	VM2		
t11	1.33	VM3		
t5	2	VM4		
t12	1.6	VM5		
t3	2	VM6		
t6	20	Wait(2)	VM1	
t9	30	Wait(2)	VM2	
t13	13.33	Wait(1.33)	VM3	
t14	16.66	Wait(2)	VM4	
t2	28	Wait(1.6)	VM5	
t15	32	Wait(2)	VM6	
t7	180	Wait(22)		VM1
t1	200	Wait(32)		VM2
t8	66.66	Wait(14.66)		VM3

SJF Work Mechanism

Advantages

1.Wait time is lower than FCFS.

2.SJF has minimum average waiting time among all tasks scheduling algorithms.

Disadvantages

1.Unfairness to some tasks when tasks are assigned to VM, due to the long tasks tending to be left waiting in the task list while small tasks are assigned to VM.

2.Taking long execution time .

MAX-MIN

In MAX-MIN tasks are sorted based on the completion time of tasks; long tasks that take more completion time have the highest priority. Then assigned to the VM with minimum overall execution time in VMs list.

MAX-MIN Work Mechanism

Tasks	t1	t8	t7	t15	t2	t14	t13	t9	t6	t3	t12	t5	t11	t10	t4
lengths	100000	100000	90000	80000	70000	25000	20000	15000	10000	5000	4000	3000	2000	1000	1000

MAX-MIN Work Mechanism

Task	ET	Waiting time		
t1	40	VM6		
t8	40	VM5		
t7	60	VM4		
t15	53.33	VM3		
t2	140	VM2		
t14	50	VM1		
t13	8	Wait(40)	VM6	
t9	6	Wait(40)	VM5	
t6	6.66	Wait(60)	VM4	
t3	3.33	Wait(53.33)	VM3	
t12	8	Wait(140)	VM2	
t5	6	Wait(50)	VM1	
t11	0.8	Wait(48)		VM6
t4	0.4	Wait(46)		VM5
t10	0.67	Wait(66.67)		VM4

MAX-MIN

Advantages

1.Working to exploit the available resources in an efficient manner.

2.This algorithm has better performance than the FCFS, SJF, and MIN-MIN algorithm.

Disadvantages

Increase waiting time to small and medium tasks; if we have six long tasks, in MAX-MIN scheduling algorithm they will take priority in six VMs in VM list, and short tasks must be waiting until the large tasks finish.

Additional References

1. https://www.intechopen.com/books/scheduling-problems-new-applications-and-trends/types-of-task-scheduling-algorithms-in-cloud-computing-environment

Distributed System: Truth is defined by majority

- A node cannot necessarily trust its own judgment of a situation.
- A distributed system cannot exclusively rely on a single node, because a node may fail at any time, potentially leaving the system stuck and unable to recover.
- Instead, *many distributed algorithms rely on a quorum*, that is, voting among the nodes :

decisions require some minimum number of votes from several nodes in order to reduce the dependence on any one particular node.

What is Distributed Locking

- When you “lock” data, you first acquire the lock, giving you exclusive access to the data. You then perform your operations. Finally, you release the lock to others.
- This sequence of acquire, operate, release is pretty well known in the context of shared-memory data structures being accessed by threads.

- With distributed locking, we have the same sort of acquire, operate, release operations, but instead of having a lock that’s only known by threads within the same process, or processes on the same machine*, use a lock, that different clients on different machines can acquire and release.*

https://redislabs.com/ebook/part-2-core-concepts/chapter-6-application-components-in-redis/6-2-distributed-locking/

Advantages of Distributed Locking

"Locking often isn't a good idea, and trying to lock something in a distributed environment may be more dangerous."

Then why would we use locks in a distributed environment?

- The purpose of a lock is to ensure that among several nodes that might try to do the same piece of work, only one actually does it (at least only one at a time).
- That work might be to write some data to a shared storage system, to perform some computation, to call some external API, or suchlike.
- *At a high level, there are two reasons why you might want a lock in a distributed application:*

Advantages of Distributed Locking

1. Efficiency:

- A lock can save software from performing useless work more times than it is really needed (e.g. some expensive computation like triggering a timer twice).
- If the lock fails and two nodes end up doing the same piece of work,

  the result is a minor increase in cost (you end up paying ₹5 more to AWS than you otherwise would have) or minor inconvenience (e.g. a user ends up getting the same email notification twice).

Advantages of Distributed Locking

2. Correctness:

- A lock can prevent the concurrent processes of the same data, avoiding data corruption, data loss, inconsistency and so on.
- If the lock fails and two nodes concurrently work on the same piece of data, the result is a corrupted file, data loss, permanent inconsistency, the wrong dose of a drug administered to a patient, or some other serious problem.

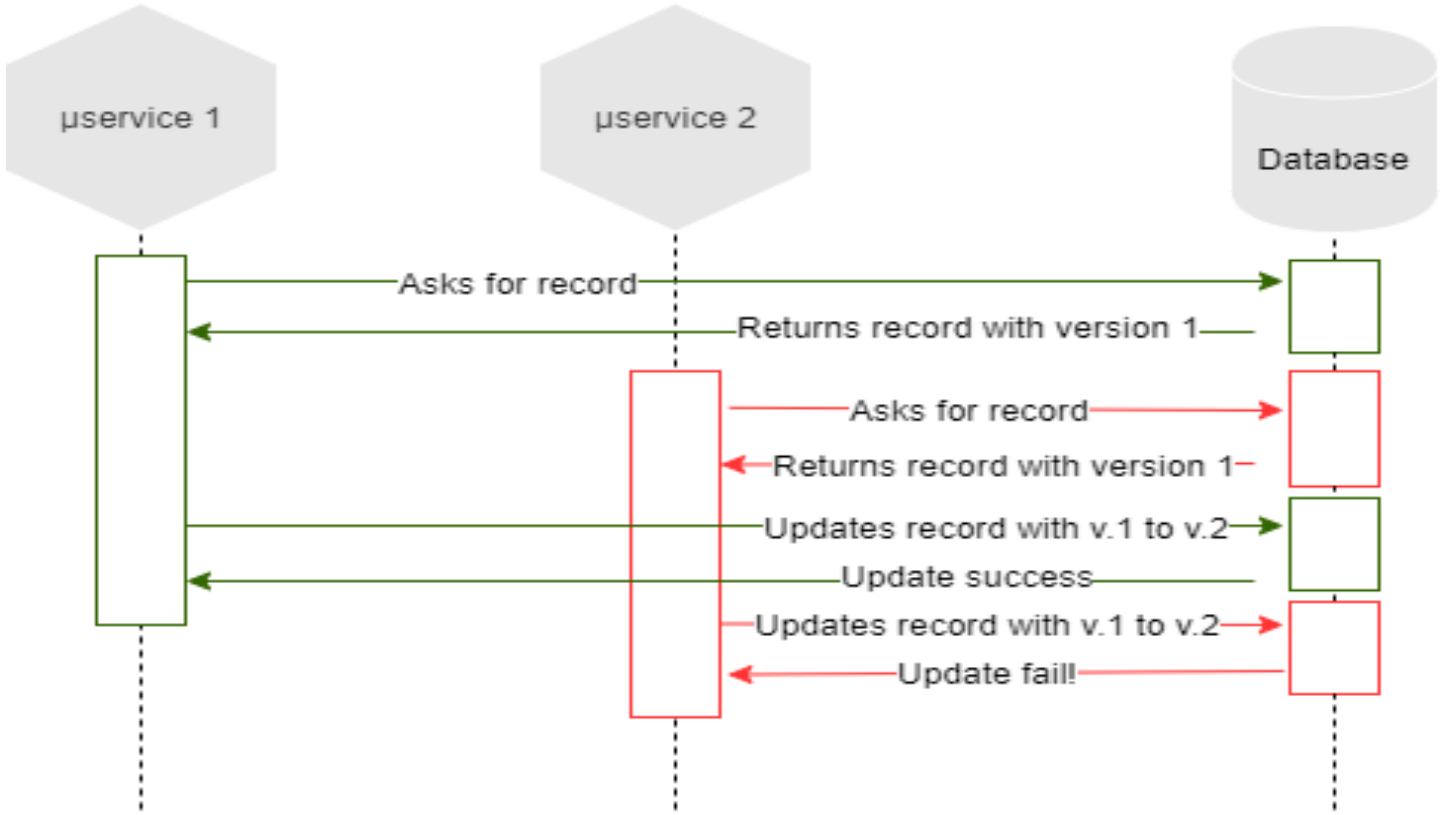
Features of distributed locks

- **Mutual Exclusion** :
  Only one client can hold a lock at a given moment.
- **Deadlock free** :
  Distributed locks use a ***lease-based locking mechanism.***
  If a client acquires a lock and encounters an exception, the lock is automatically released after a certain period. This prevents resource deadlocks.
- **Consistency** :
  Failovers may be triggered by external errors or internal errors.
  External errors include hardware failures and network exceptions, and internal errors include slow queries and system defects.
  In many implementations, after failovers are triggered, a replica serves as a master to implement high availability.
  In this scenario, if services have high requirements for mutual exclusion, the lock of the client must remain the same after the failovers.

https://www.alibabacloud.com/help/doc-detail/146758.htm

CLOUD COMPUTING

Optimistic Locks



Optimistic lock sequence diagram

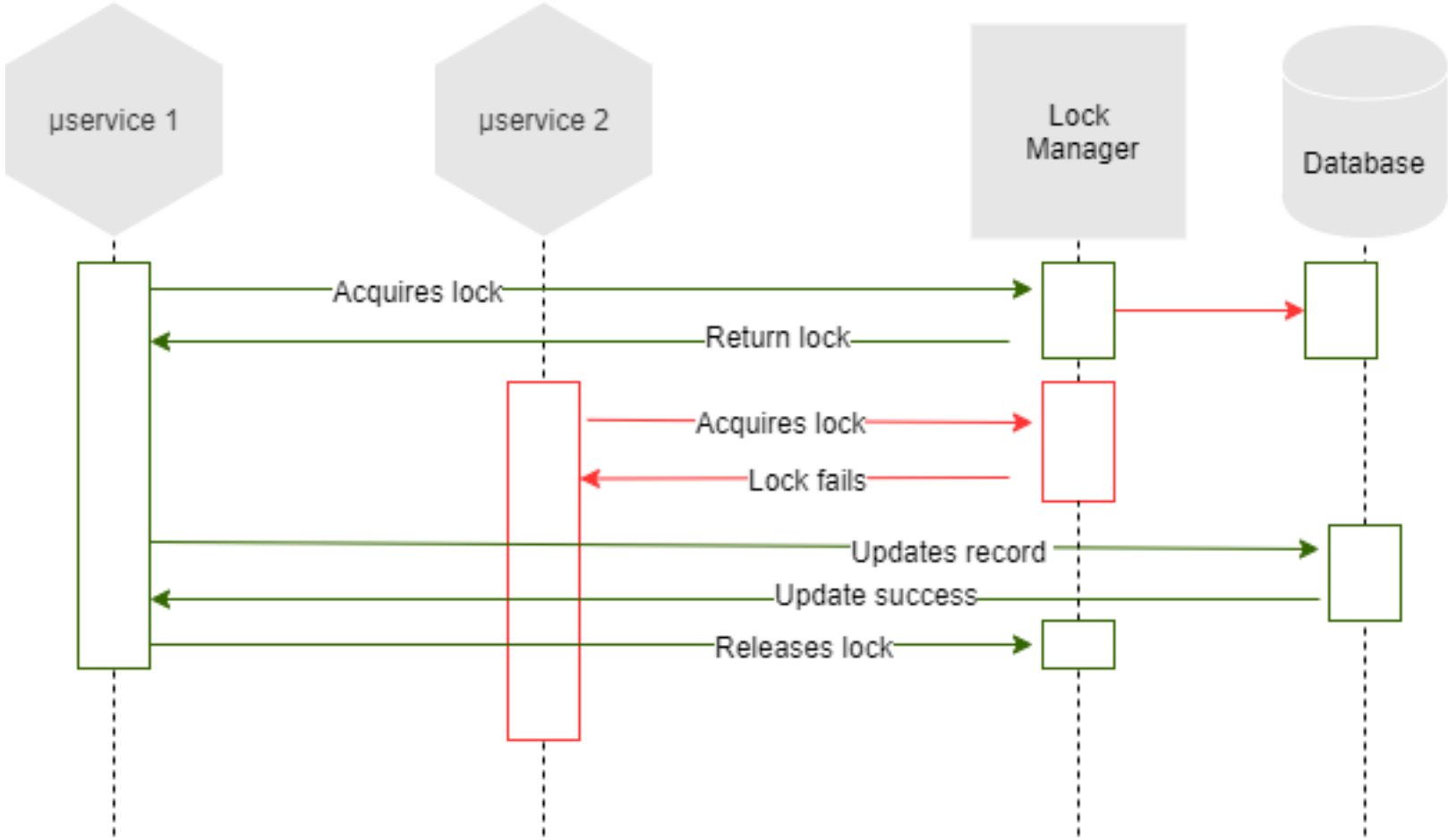
Optimistic:

Instead of blocking something potentially dangerous happens, we continue anyway, in the hope that everything will be ok.

https://dzone.com/articles/everything-i-know-about-distributed-locks

CLOUD COMPUTING

Pessimistic Locks



Pessimistic lock sequence diagram

Pessimistic: Block access to the resource before operating on it, and we release the lock at the end.

https://dzone.com/articles/everything-i-know-about-distributed-locks

Implementing Distributed Locking

- It's important to remember that a lock in a distributed system is not like a mutex in a multi-threaded application.
- It's more complicated, due to the problem that different nodes and the network can all fail independently in various ways.

For example, say you have an application in which a client needs to update a file in shared storage (e.g. HDFS or S3).

- A client first acquires the lock, then reads the file, makes some changes, writes the modified file back, and finally releases the lock.
- The lock prevents two clients from performing this read-modify-write cycle concurrently, which would result in lost updates.
- A code that might be written to perform this is shown on the next slide.
- This however can still lead to corrupted data.

https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html

CLOUD COMPUTING

Implementing Distributed Locking

```
// 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();
    }
}
```

https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html

CLOUD COMPUTING

Implementing Distributed Locking

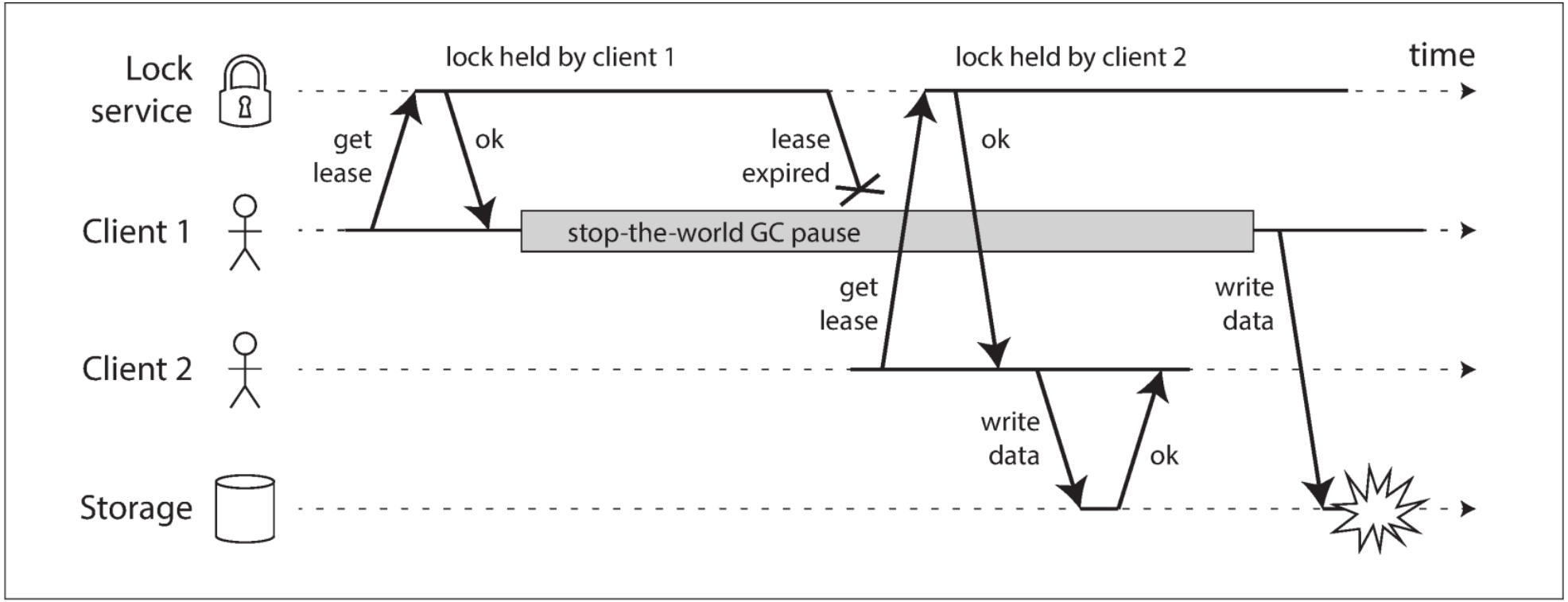


Figure 8-4. Incorrect implementation of a distributed lock: client 1 believes that it still has a valid lease, even though it has expired, and thus corrupts a file in storage.

https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html

Implementing Distributed Locking

In this example,

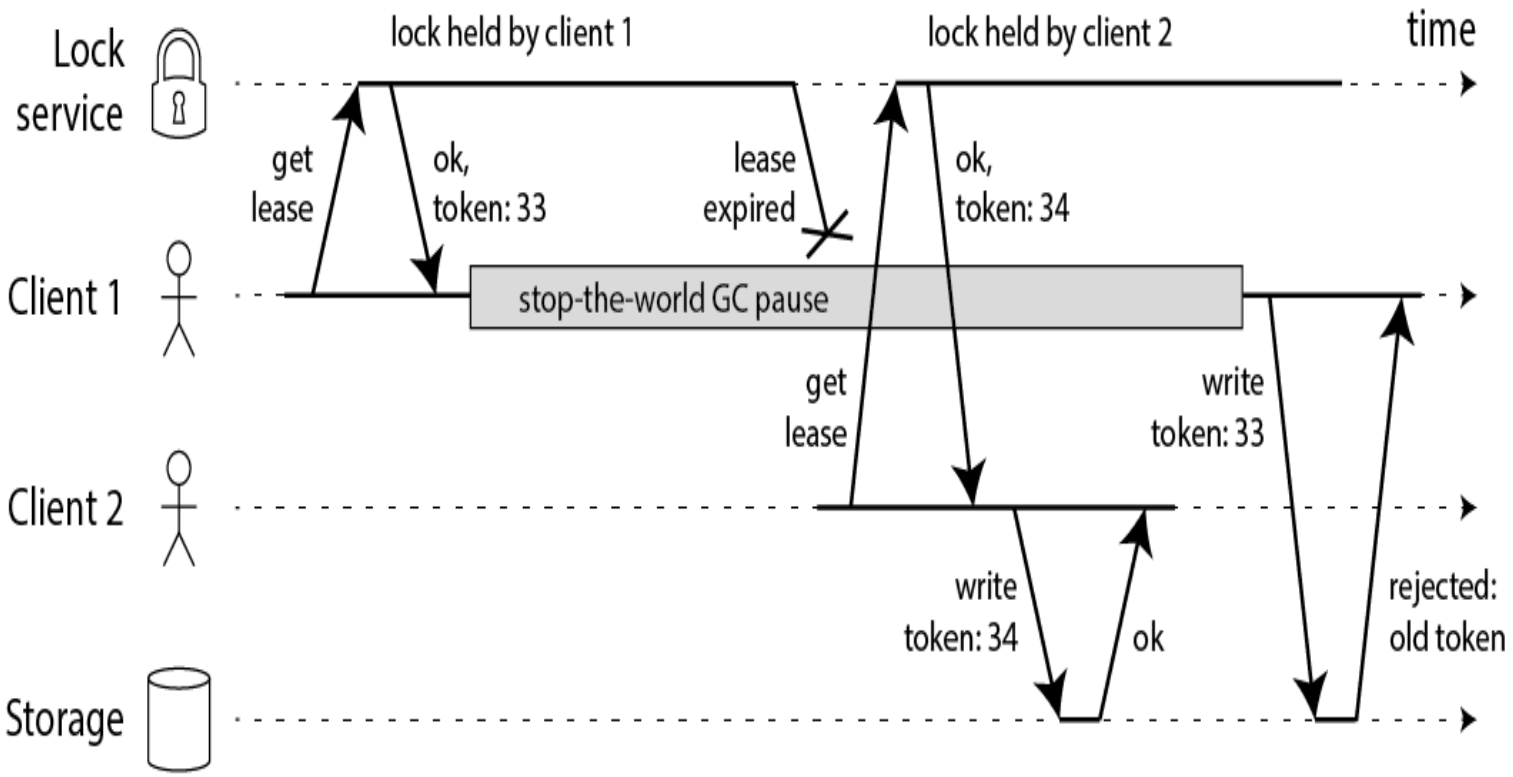
- the client that acquired the lock is paused for an extended period of time while holding the lock
- for example because the garbage collector kicked in.
- The lock has a timeout (i.e. it is a lease), which is always a good idea (otherwise a crashed client could end up holding a lock forever and never releasing it).
- However, if the GC pause lasts longer than the lease expiry period, and the client doesn't realise that it has expired, it may go ahead and make some unsafe change.

NOTE: GC stands for Garbage Collector

https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html

CLOUD COMPUTING

Implementing Distributed Locking with Fencing



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 of being "the chosen one" cannot disrupt the rest of the system.

NOTE: GC stands for Garbage Collector

https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html

Implementing Distributed Locking with Fencing

- 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.

https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html

CLOUD COMPUTING

Implementing Distributed Locking with Fencing

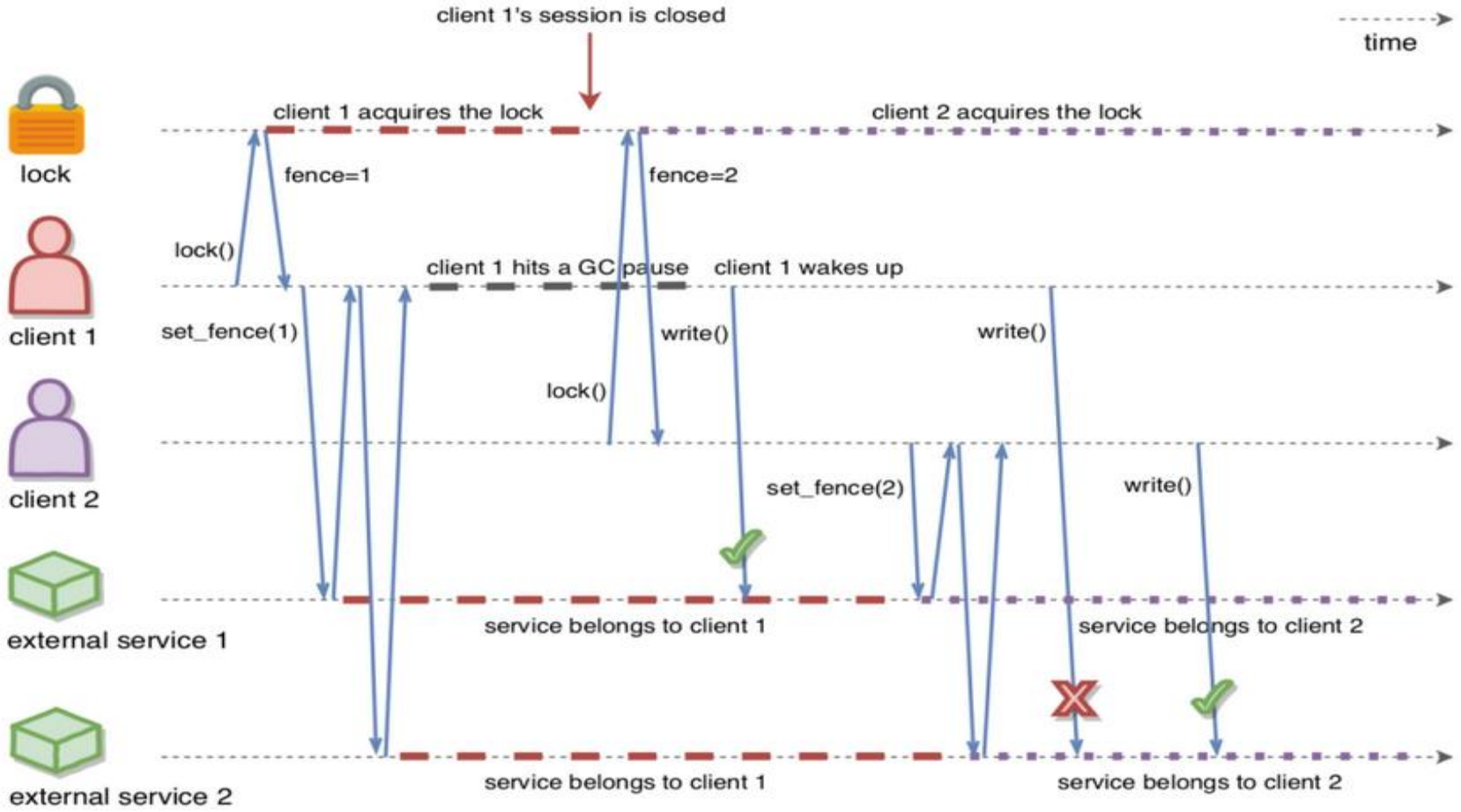


Figure 1: Using fencing tokens to fence off stale lock holders

https://hazelcast.com/blog/long-live-distributed-locks/

Distributed Lock Manager

A distributed lock manager (DLM)

- runs in every machine in a cluster, with an identical copy of a cluster-wide lock database.
- In this way a DLM provides software applications which are distributed across a cluster on multiple machines with a means to synchronize their accesses to shared resources.
- The DLM uses a generalized concept of a resource, which is some entity to which shared access must be controlled.
- This can relate to a file, a record, an area of shared memory, or anything else that the application designer chooses.

https://en.wikipedia.org/wiki/Distributed_lock_manager

Distributed Lock Manager

DLM's can be implemented using the following:

- Google has developed ***Chubby***, *a lock service for loosely coupled distributed systems.*
- ***Apache ZooKeeper*** *is open-source software and can be used to perform distributed locks.*
- ***Redis*** is an open source, advanced key-value cache and store. Redis can be used to implement the Redlock Algorithm for distributed lock management.

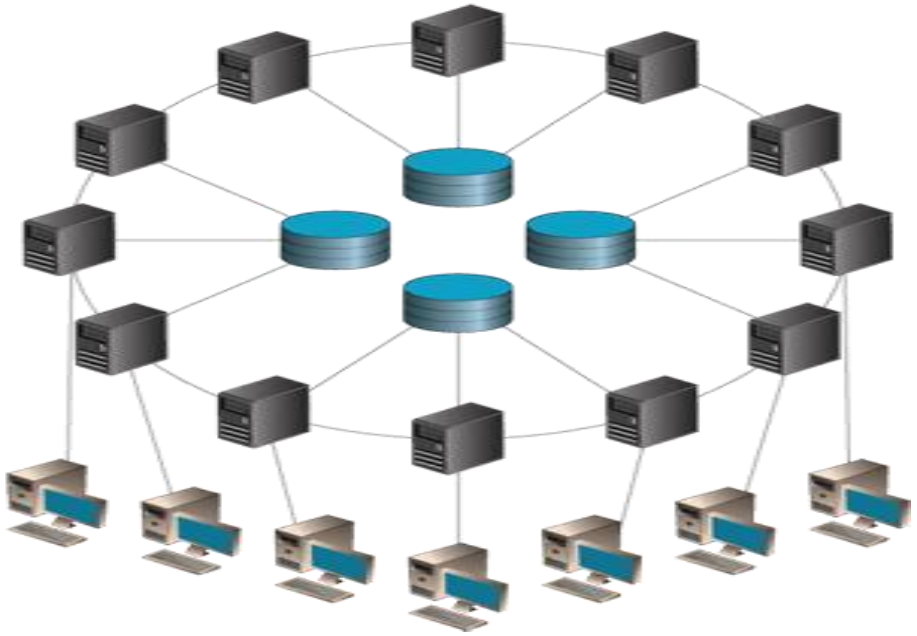
https://en.wikipedia.org/wiki/Distributed_lock_manager

Further Reading

- Distributed Locks are Dead; Long Live Distributed Locks!
- Distributed Lock using Zookeeper
- Distributed locks with Redis

Why Zookeeper is needed?

- Distributed systems always need some form of coordination
- Programmers cannot use locks correctly
- Message based coordination can be hard to use in some applications



Apache Zookeeper

What is Apache Zookeeper?

- Open source distributed coordination service that helps to manage a large set of hosts while

  Management and coordination in a distributed environment is tricky.
- Automates this process and allows developers to focus on building software features rather than worry about it's distributed nature.
- ***Centralized coordination service that can be used by distributed applications to maintain configuration information, perform distributed synchronization, and enable group services.***
- Apache Kafka uses ZooKeeper to manage configuration, Apache HBase uses ZooKeeper to track the status of distributed data.

CLOUD COMPUTING

Apache Zookeeper Features

- Offers a hierarchical key-value store, to provide a
  distributed configuration service,
  synchronization service, and
  naming registry

  for large distributed systems

Apache Zookeeper Features

1. **Updating the Node's Status**: updates every node that allows it to store updated information about each node across the cluster.
2. **Managing the Cluster**: manage the cluster in such a way that the status of each node is maintained in real time, leaving lesser chances for errors and ambiguity.
3. **Naming Service**: attaches a unique identification to every node which is quite similar to the DNA that helps identify it.
4. **Automatic Failure Recovery**: locks the data while modifying which helps the cluster recover it automatically if a failure occurs in the database.

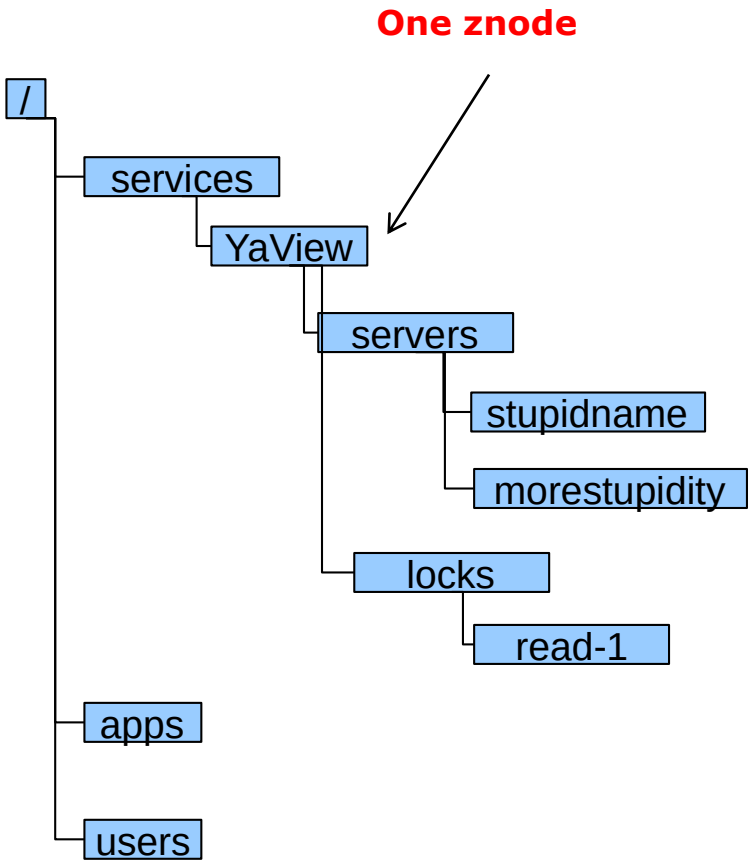
Apache Zookeeper: Objectives

- Simple, Robust, Good Performance
- High Availability, High throughput, Low Latency
- Tuned for Read dominant workloads
- Familiar models and interfaces
- Wait-Free: A slow/failed client will not interfere with the requests of a fast client

Apache Zookeeper: Data Model

- Hierarchal namespace (like a file system)
- Each node is called "***znode***"
- Each ***znode*** has data and children

The entire Zookeeper tree is kept in main memory



How does Zookeeper work?

- ZooKeeper uses a shared hierarchical name space of data registers (called znodes) to coordinate distributed processes.
- Znodes give an abstraction of a shared file system but are more like a distributed, consistent shared memory that is hierarchically organized like a file system.
- For reliability, three copies of the Zookeeper can be run so that it does not become a single point of failure.

How does Zookeeper work?

- ZooKeeper provides a name space very similar to a standard file system.
- Every znode is identified by a name, which is a sequence of path elements separated by a slash ("/"), and every znode has a parent except the root ("/").
- A znode cannot be deleted if it has any children.
- Every znode can have data associated with it and is limited to the amount of data that it can have to kilobytes.
- This is because the ZooKeeper is designed to store just coordination data, such as status information, configuration, location information, and so on.

How does Zookeeper work?

- Maintain a stat structure that includes version numbers for data changes, acl changes and timestamps
- When a client performs an update or a delete, it must supply the version of the data of the znode it is changing.
- If the version it supplies doesn't match the actual version of the data, the update will fail

Apache Zookeeper: What is stored in znodes?

- Configurations
- Status
- Counters
- Parameters
- Location information

- **Keeps metadata information**
- **Does not store big datasets**

Apache Zookeeper: Types of Znodes

Persistent znodes

- Permanent and have to be deleted explicitly by the client.
- Stay even after the session that created the znode is terminated.

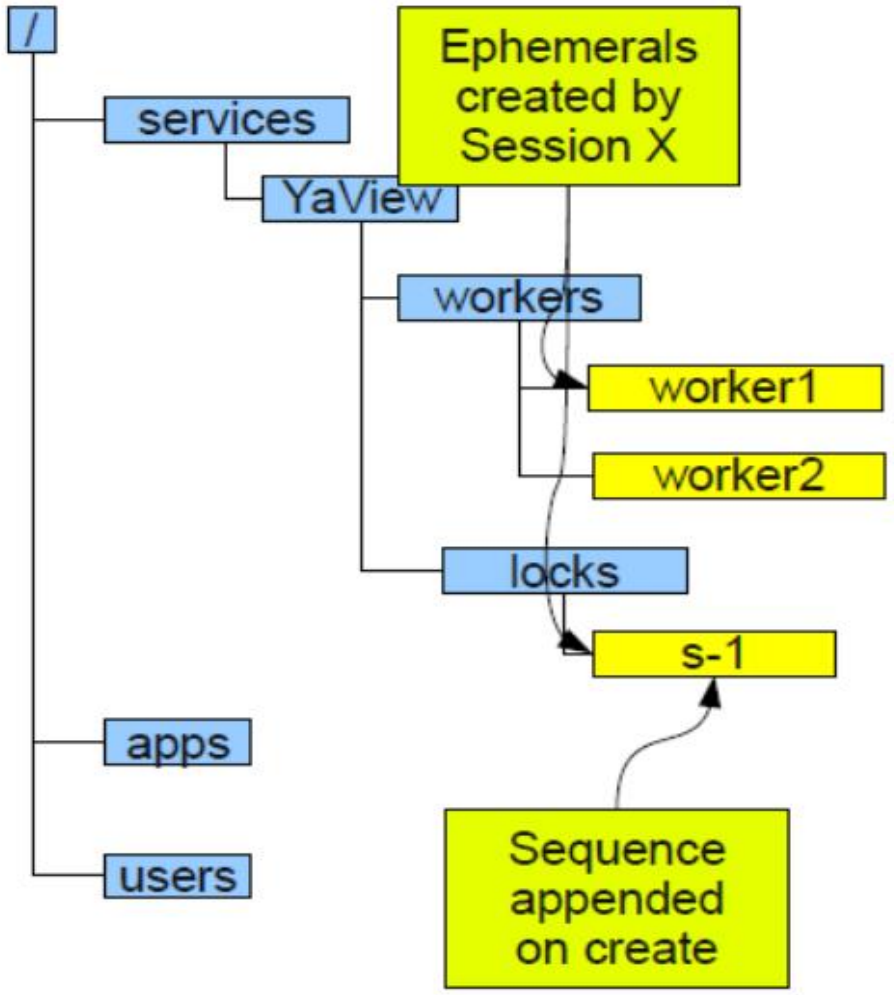
Ephemeral znodes

- Is deleted by ZooKeeper service when the creating client's session ends
- Can also be explicitly deleted
- Are not allowed to have children

Sequential- Is assigned a sequence number by ZooKeeper as a part of name during creation . Sequence number is integer (4bytes) with format of 10 digits with 0 padding. E.g. /path/to/znode-0000000001

Apache Zookeeper: Data Model

- Watches
- Data Access- Access Control List (ACL) nodes
- Flags
  - Ephemeral
  - Sequential



Apache Zookeeper: Data Model

Watches

- One time trigger, Changes to znode trigger the watch and then clear the watch

Data Access

- The data stored at each znode in a namespace is read and written atomically. Each node has an Access Control List (ACL) that restricts who can do what.

Ephemeral Nodes

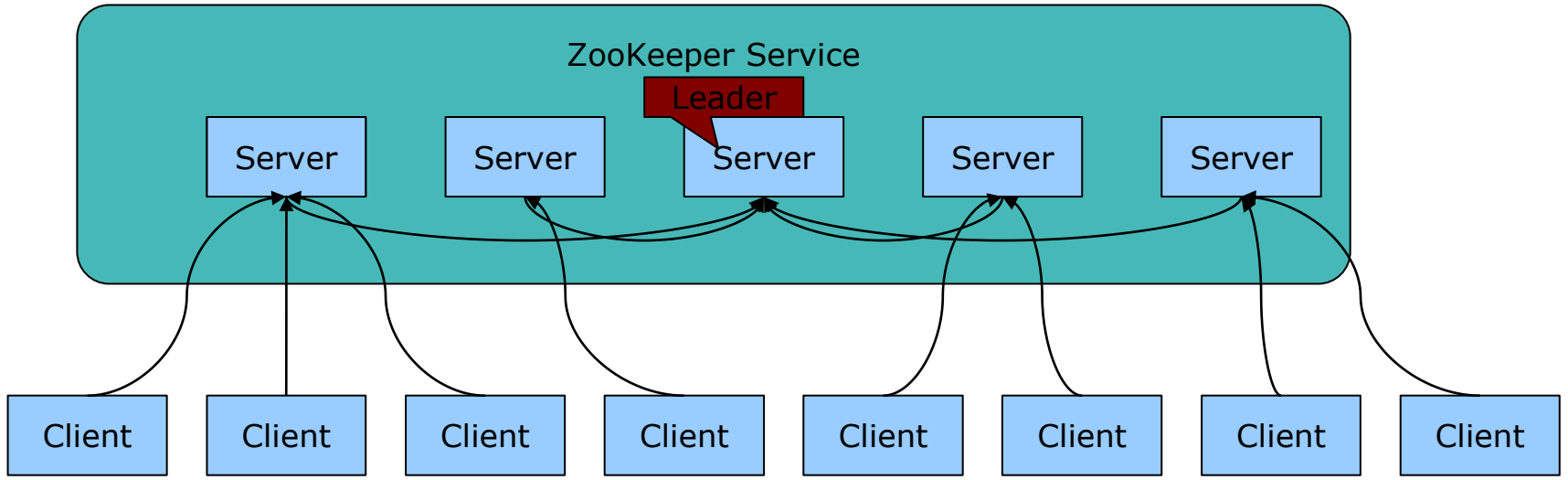
- Exists as long as the session that created the znode is active. When the session ends the znode is deleted. Because of this behavior ephemeral znodes are not allowed to have children.

Sequence Nodes -- Unique Naming

- When creating a znode you can also request that ZooKeeper append a monotonicly increasing counter to the end of path.

  This counter is unique to the parent znode

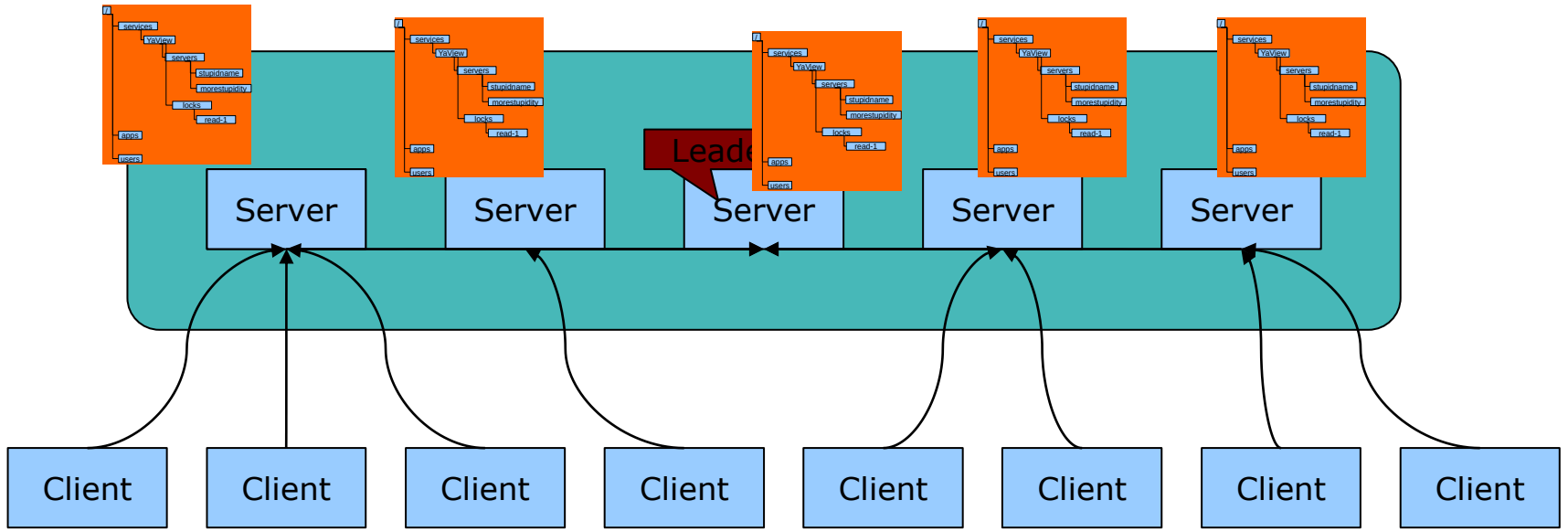
Zookeeper Servers



- A leader is elected at startup
- Followers service clients
- A client establish a connection with one server

Zookeeper Servers

All servers store a copy of the data tree (in memory)



- *Simple ,Robust, Good performance*
- *High Availability, throughput and low latency*
- *Wait-free – A slow/failed client does not interfere with the requests of a fast client*

Zookeeper

- The ZooKeeper service maintains an in-memory image of the data tree that is replicated on all the servers on which the ZooKeeper service executes.

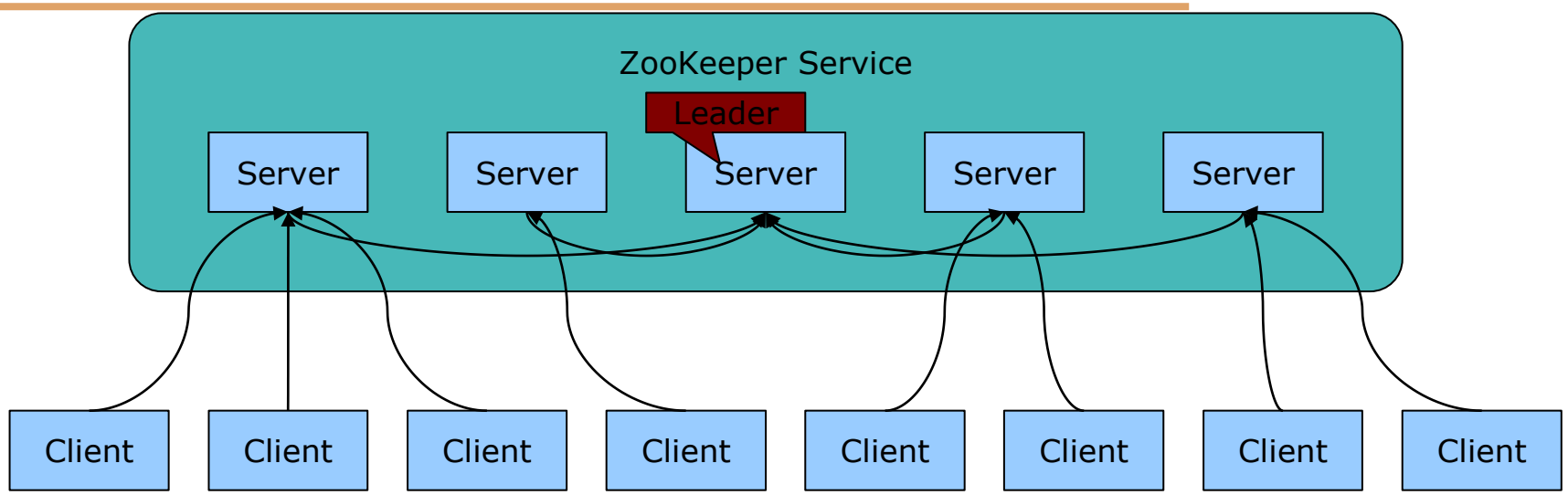
- Only *transaction logs and snapshots* are stored in a *persistent store*, enabling high throughput. As changes are made to the znodes (during application execution), they are appended to the transaction logs.

- Each client connects only to a single ZooKeeper server and maintains a TCP connection, through which it sends requests, gets responses, gets watch events, and sends heart beats. If the TCP connection to the server breaks, the client will connect to an alternate server.

Zookeeper

- All updates made by ZooKeeper are totally ordered. It stamps each update with a sequence called zxid (ZooKeeper Transaction Id). Each update will have a unique zxid.
- To achieve distributed synchronization, ZooKeeper implements a variant of the classic part time parliament or Synod protocol called the Zab (ZooKeeper Atomic Broadcast) protocol.
- The clients just read and write files.
- Read requests from a client are processed locally at the ZooKeeper server to which the client is connected, while write requests are forwarded to other ZooKeeper servers.

Zookeeper Reads/Writes/Watches



- **Reads:** between the client and single server
- **Writes:** sent between servers first, get consensus, then respond back
- **Watches:** between the client and single server
  - A watch on a znode basically means *"monitoring it"*

Zookeeper API

One of the design goals of ZooKeeper is providing a very simple programming interface.

- *create* : creates a node at a location in the tree
- *delete* : deletes a node
- *exists* : tests if a node exists at a location
- *get data* : reads the data from a node
- *set data* : writes data to a node
- *get children* : retrieves a list of children of a node
- *sync* : waits for data to be propagated

Zookeeper API

String create(path, data, ACL, flags)

void delete(path, expectedVersion)

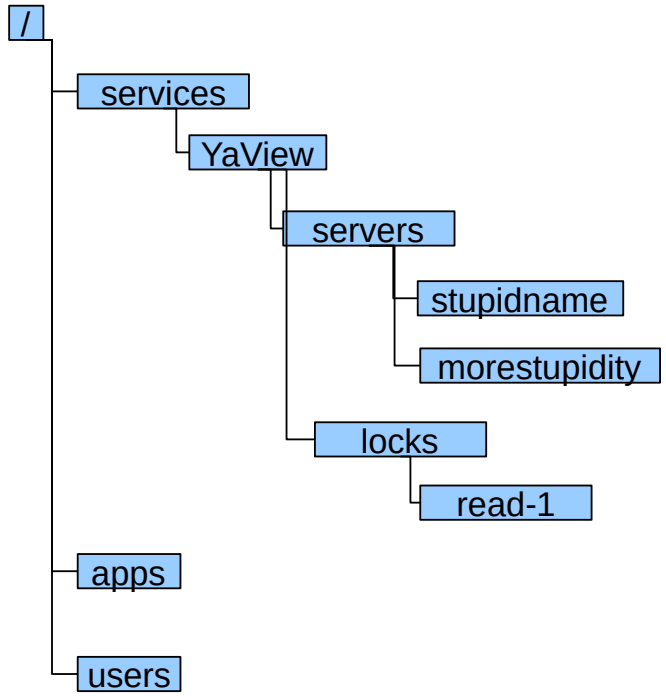
Stat setData(path, data, expectedVersion)

(data, Stat) getData(path, watch)

Stat exists(path, watch)

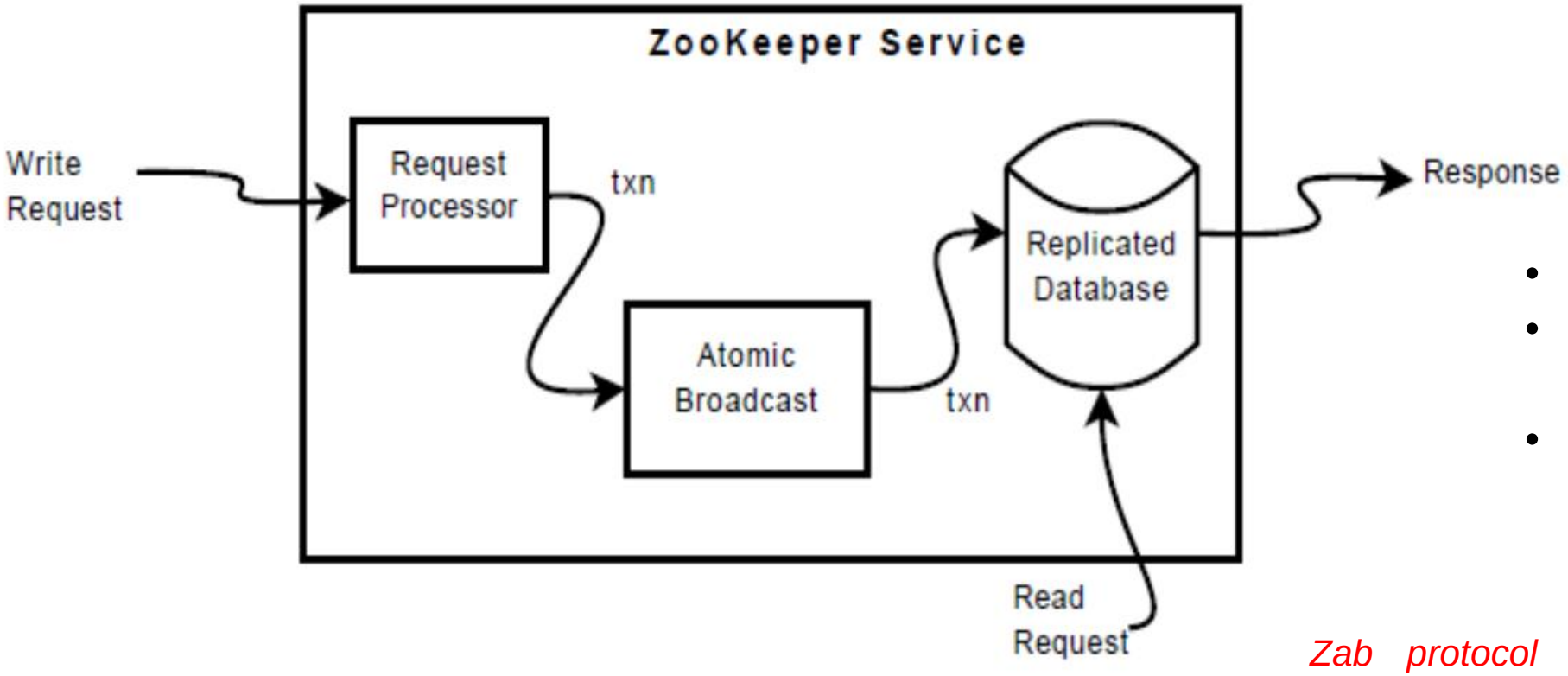
String[] getChildren(path, watch)

…



CLOUD COMPUTING

Implementation of Zookeeper



- Request Processor
- Atomic Broadcast
  - Zab Protocol
- Replicated Database

Zab protocol ensures that the Zookeeper replication is done in order and is also responsible for leader node elections and restoring failed nodes if any.

Advantages of Zookeeper(Guaranty Consistency)

- **Simple distributed coordination process, Sequential consistency** (client updates are done in the order of the requests)
- **Synchronization** – Mutual exclusion and co-operation between server processes. This process helps in Apache HBase for configuration management.
- **Single system image** – Client sees same view of Zookeeper service regardless of server
- **Ordered Messages**
- **Serialization** – Encode the data according to specific rules. Ensure your application runs consistently. This approach can be used in MapReduce to coordinate queue to execute running threads.
- **Reliability**
- **Atomicity** – Data transfer either succeeds or fails completely, no transaction is partial.

https://www.tutorialspoint.com/zookeeper/zookeeper_overview.htm

Disadvantages of Zookeeper

- Other solutions like consul have features like service discovery baked in, with health checks in the loop, while Zookeeper does not.
- The ephemeral node lifecycle is tied to TCP connection; it can happen that TCP connection is up, but the process on the other side just went out of memory or otherwise not functioning properly
- It’s fairly Complex

Consul provides many different **features** that are used to provide consistent and available information about your infrastructure. This includes service and node discovery mechanisms, a tagging system, health checks, consensus-based election routines, system-wide key/value storage, and more

So, for what should I use Zookeeper?

- **Naming service** - Naming Service is a specific service whose aim is to provide a consistent and uniform naming of resources, thus allowing other programs or services to localize them and obtain the required metadata for interacting with them
- **Configuration management**
- **Data Synchronization** - Data synchronization is the process of establishing consistency among data from a source to a target data storage and vice versa and the continuous harmonization of the data over time.

Zookeeper Use Cases

So, for what should I use Zookeeper?

- **Leader election** - Leader election is the process of designating a single process as the organizer of some task distributed among several computers.
- **Message queue** - Typically used for inter-process communication, or for inter-thread communication within the same process.
- **Distributed Locks**

Who uses ZooKeeper?

Companies:

- Yahoo!
- Zynga
- Rackspace
- Linkedlin
- Netflix, and many more...

Projects:

- Apache Map/Reduce (Yarn)
- Apache HBase
- Apache Kafka
- Apache Storm
- Neo4j, and many more...

THANK YOU

Ms. Suganthi S
Mail-id: suganthis@pes.edu

Department of Computer Science and Engineering