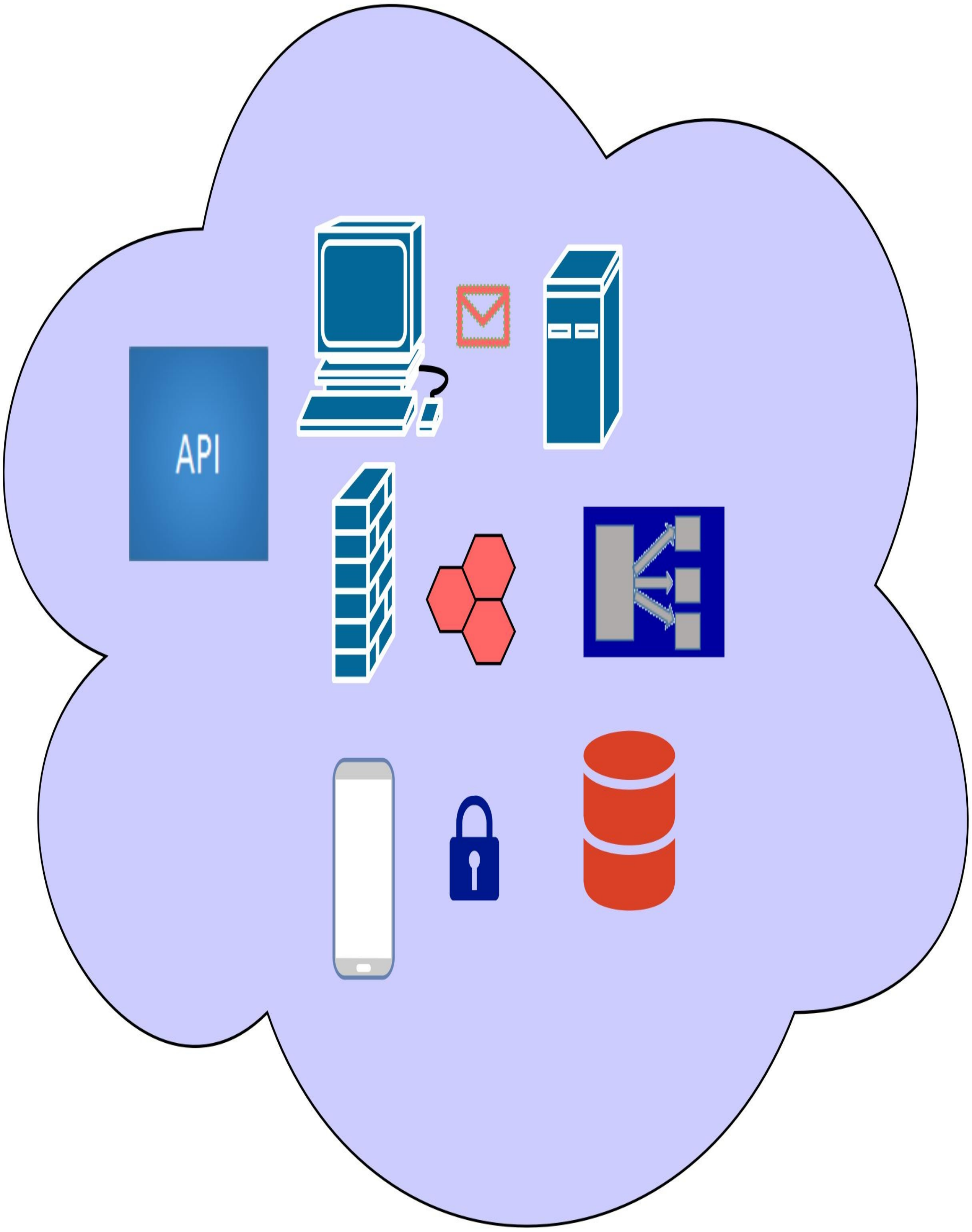
A Guide to Software Architecture



- Raghuveer Bhandarkar

Table of Contents

A Guide to Software Architecture

This book introduces modern Software Architecture concepts by covering a wide range of topics from Infrastructure, Application Architecture, and Data Architecture. The topics are very relevant to the current industry trends with a focus on Cloud and Distributed computing.

This book would serve as a quick handy guide for Solution Architects. Emphasis is given to architecture concepts keeping distributed computing in mind which is an important factor in designing systems in cloud.

Each concept is illustrated by easy to follow diagrams and is backed up with reference implementations.

The audience for this book could be experienced Software engineers, aspiring to-be Architects, practicing Software Architects, and Consultants working on Cloud. The reader of this book would get to know entire spectrum of Solution Architecture.

Preface

I have been writing this book for more than a year now. The idea to write this book arouse when a lot of my friends and colleagues frequently asked me 'There are so many changes going on in technology at a rapid pace and what is the best book available which would cover all of the contemporary architectural trends and concepts?'.

In today’s world, it is very essential to have a 360 degree view of the entire technology architecture space and then as and when required, the person can dive deep into specific topics. Often, an architect has to make or enable technological decisions involving data, tools, frameworks, language etc. Getting a bird’s view of the entire landscape and knowledge of various choices at hand, assist in making right decisions.

For instance, an architect may not be an expert in infrastructure as other infrastructure folks, but has to have a sufficient high level knowledge to come up with system design and interact with all stakeholders.

I tried to find a book for myself which would serve the above purpose, but couldn’t find the right book. I feel this book best serves the above purpose. This book provides end to end knowledge to design and architect an application in a Cloud and Distributed environment.

About the Author

The author is a Senior Solutions Architect working for a MNC and has over 14 years of experience in Software industry with expertise in designing and architecting applications. The author is passionate about technology, a technology evangelist and a vivid blogger.

Acknowledgements

I would like to dedicate this book to my parents who have been a constant source of inspiration to me. I would also like to thank the readers in advance for devoting their time in reading this book.

Table of Contents

General Architecture

This section explains the various general concepts in Software Architecture in detail.

Non Functional Requirements

Functional requirements of a system define how a system should behave with specific use cases. An example of a functional requirement could be “when the user clicks Submit button, the order should be placed”. Non Functional requirements, on the other hand, define the operational quality of a system. They define criteria or parameters which are used to measure the operational efficiency of an application.

The non-Functional requirements need not be explicitly defined, but an architect needs to consider these while designing the system.

Few of the common Non Functional Requirements are listed out below:

1. System Performance (considering Speed and/or memory):

This requirement concerns about how the system performs during peak loads. A few pointers on this are provided below:

- How much memory will it consume during peak load?
- Will the infrastructure/hardware support such peak traffic?
- How many transactions per second can the system support?

2. System Security:

Security encompasses wide ranging concerns as below:

- Data security (including both data in motion and data at rest)
- Security of the web layer against various attacks
- Security of the infrastructure components
- Application user security (with respect to User roles)
- Security processes like SSL certificate/Key policy, Renewal processes, maintenance of certificates/Keys.

3. Usability:

Usability, concerns about the ease of use of the system, from User Interface perspective. There could be certain tradeoffs between usability and performance or usability and security. For example a certain feature could be extremely user friendly but could impact performance or compromise security.

4. Availability:

Availability, concerns with the average up time of the system. For e.g., if the requirement mandates that the system be available ‘24X7X365’, then it implies that the system be available all the times. Also, we need to factor in the downtimes during upgrades or release times, if required. This in turn, could decide the deployment/upgrade strategies.

5. Compliance:

This requirement concerns with compliance to laws and regulations depending on local laws, State/Country specific laws. For e.g., certain country laws do not allow personal data to reside outside of the country. So if we are considering using a public cloud (which could host data in any datacenter across the world) for a system which collects personal data from citizens, then we need check the regulations for that country.

6. Fault tolerance:

Fault tolerance, concerns with the system behavior when there is a failure. The failure could be a hardware failure, or failure of an application component or a failure of installed software or even a malfunction of the application. When these failures happen, we need to understand what is the impact on the end user and how is the normal functioning of the system impacted. Also, we need to figure out ways to recover from failures, and how to complete or retry any incomplete or erroneous transactions. These factors would have a major impact on the system design. For instance, if the system is interacting with another third party system, then we need to account for failures within our system and also the possible impact on the third party system as well. We should also look out for Single point of failures within the system (This concept is explained in detail in a separate section, later in the book).

7. Scalability:

Scalability, concerns with the ability of the system to cater to varying workload. A few factors regarding scalability are listed out below:

- Is the system capable of scaling horizontally?
- Is every component of the system scalable?
- Is the infrastructure scalable?
- Will the network be able to cater to the peak workload?

8. Extensibility:

Extensibility, concerns with the flexibility of the system for future extensions. For example, there could be integrations with new systems in future or there could be certain requirements which would require building upon existing features without doing a lot of rework. It is important to understand the overall vision from business perspective and foresee what is coming, to be able to build an extensible system.

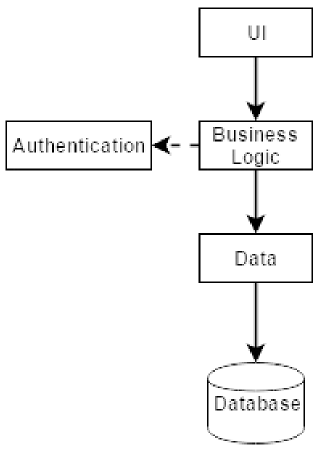
9. Maintainability:

Maintainability, concerns with the ease of maintaining and supporting the application from an operational stand point. It includes adhering to standards and using the right design patterns to ensure that the system is built in a uniform way which could be easily understood by anybody.

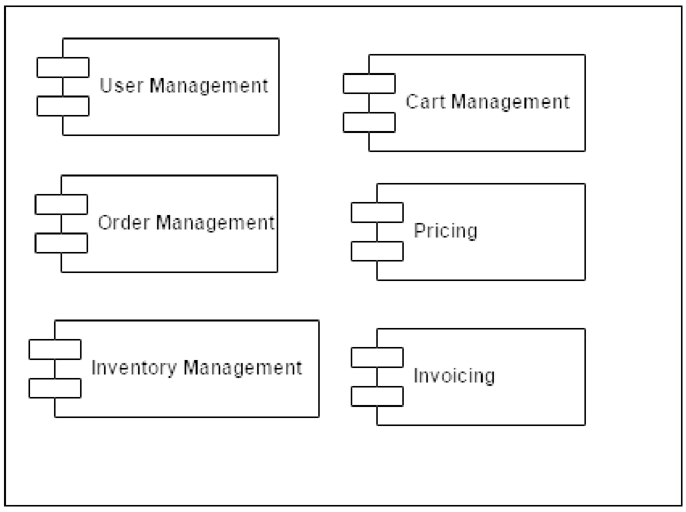
10. Interoperability:

Interoperability, concerns with the ability of the system to integrate with other systems. This is possible when we stick to open standards and open protocols. Conformance to open standards while designing (for example using standards like REST, SOAP), or using open exchange formats like XML, JSON etc. will help greatly in making the system interoperable. In certain situations, depending on the domain, we might need to use standard message formats, like standard XSDs (For example there are certain industry standard XML formats for Banking, Health care domains). Adhering to these standards help in the ease of integration with other systems.

View Points

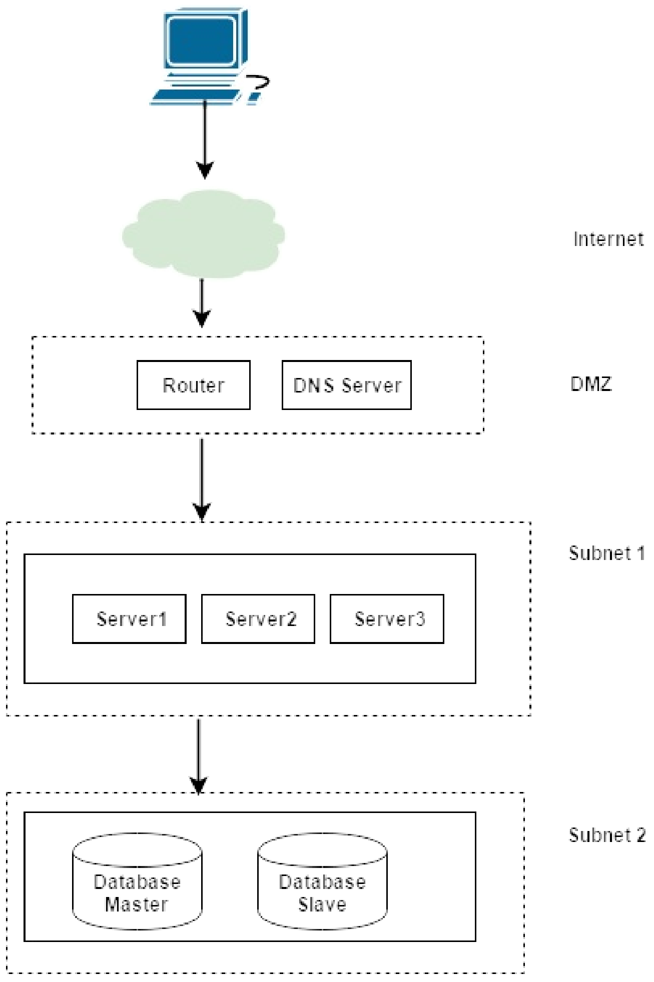


Developer View Point 1 - Layers



Developer View Point 2 - Modules

View Points



Network View Point

An architect would have to often communicate to people with different backgrounds, effectively. Presenting architecture to stakeholders with different backgrounds and getting their by-ins, could be a challenging task.

Diagrams are a very effective tool to express or communicate and it is often said that a picture is worth thousand words.

From an architecture stand point, it is crucial to present the architecture depending on the audience's perspective. For e.g., a Network administrator would understand or appreciate the architecture, if it is presented in terms of Firewalls, subnetworks, IP range etc. Similarly, a database expert would understand architecture in terms of data security, data flow, data storage options etc. Presenting architecture in different perspectives is referred to as 'View Points'.

The above diagram shows view points from a developer view (in terms of layers and modules) and network view point in terms of servers.

Some of the common View Points are outlined below:

1. Physical Architecture
2. Application Architecture
3. Deployment Architecture
4. Data Architecture

Performance Measures

In this section, we will look at three performance measures which are commonly used as a criteria in System design.

1. IOPS
2. TPS
3. Availability (in nines)

IOPS:

IOPS defined as 'Input Output operations per second' is a performance metric for measuring the rate of data transfer. This is generally used as a criterion to measure data transfer in storage devices and network devices. In any system, the overall performance is related to or affected by, the rate of data transfer in low level physical devices. Generally IOPS is measured in MB/sec or GB/sec.

When we are selecting storage device for a system, one of the criteria could be selecting a device which offers higher IOPS.

TPS:

TPS defined as 'Transactions per second' is a performance metric for measuring the amount of atomic transactions which the system can handle. It provides a measure of the traffic that the system can handle. It could be measured as an average or peak traffic value.

To calculate TPS, we would need to take into account the capacity of the application server, the database and any other system involved in an end to end transaction.

Availability (in nines):

Availability of a system refers to a state of the system wherein it successfully responds to requests. It represents a SLA (Service Level Agreement), which is guaranteed by the system.

Availability of a system could be represented as uptime or as downtime.

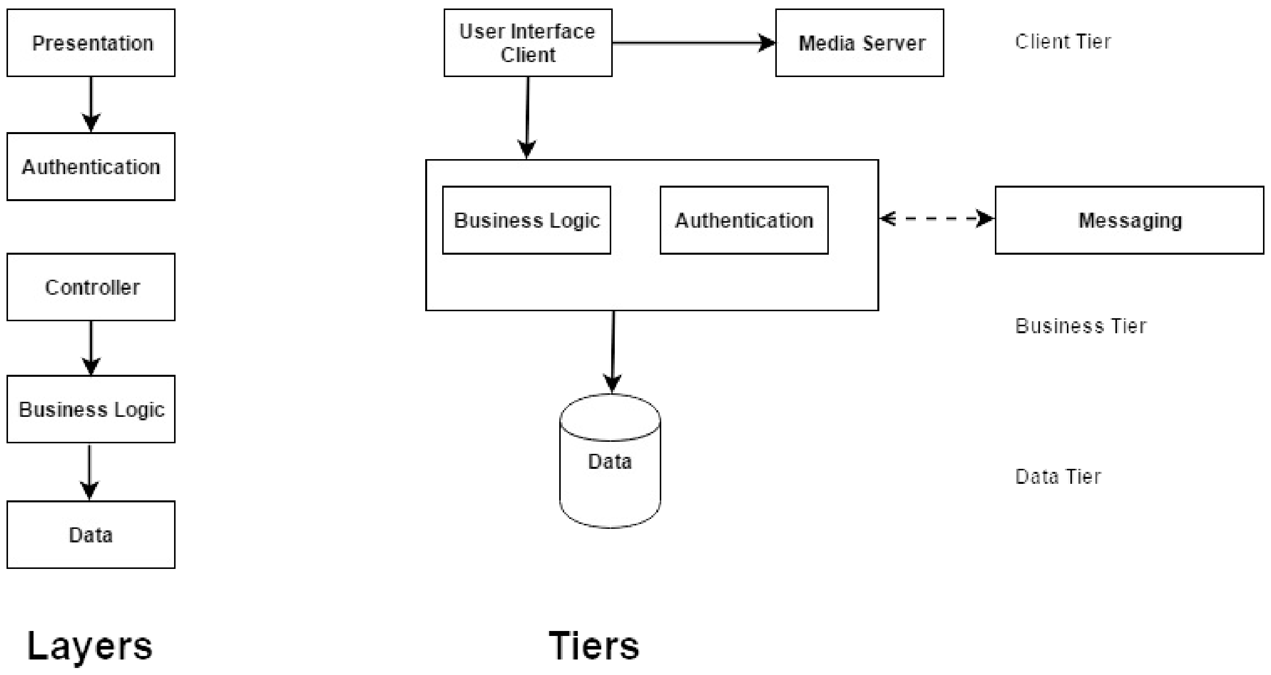
Uptime is represented as a percentage of the time duration in a year for which the system is Up or available.

Downtime is represented as a percentage of the time duration in a year for which the system is down or unavailable.

Thus, if a system is up for 90% in a year, then the downtime period for the system in one year is 10% or 36.5 days. For a month, the downtime is 72 hours. 90% uptime is also known as "One Nine".

On similar terms, 99% uptime is known as "Two nines", 99.9% is known as "Three nines" and so on.

Layers and Tiers



In Software Architecture, we often come across the terms Layers and Tiers. Layers are logical entities, whereas Ties are physical entities.

Layers:

Layers define how the code is logically organized or grouped. The above diagram shows various layers in a web application.

Examples:

1. Presentation Layer – is the layer which consists of UI logic.
2. Business Layer– is the layer which consists of business logic.
3. Data Layer– is the layer which consists of data.

Tiers:

Tiers define how the system is organized physically. For e.g., it can define how are the different components of the system located or deployed. The above diagram shows various tiers in a web application.

Examples:

1. Client Tier
2. Data Tier

3. Messaging Tier

Importance of Layers:

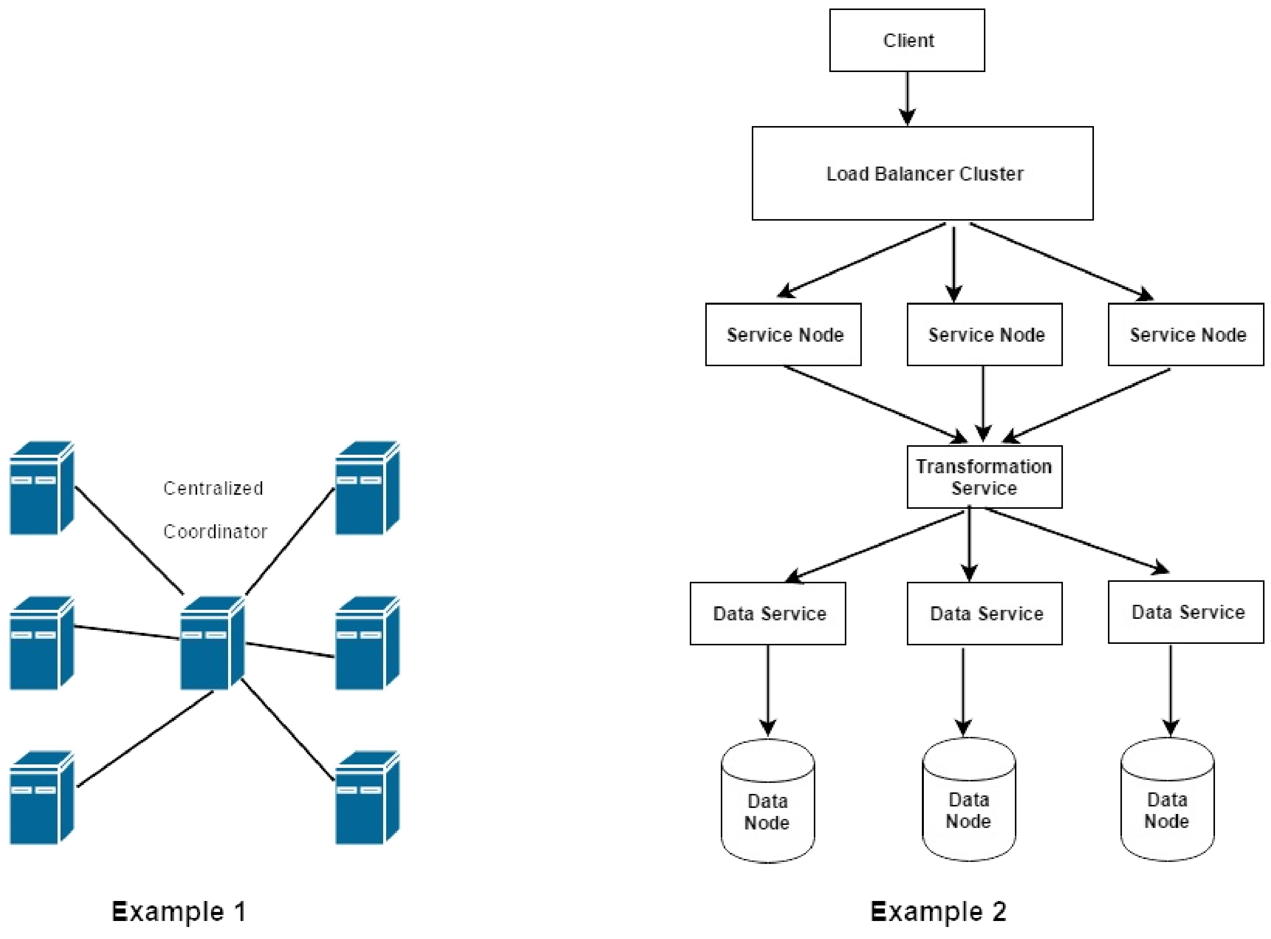
Defining layers is important in order to have code maintainability, separation of concerns and in defining the technology stack. For instance, Presentation and Business Layers might be deployed on the same machine (i.e. thereby they belong to the same tier). However, these two layers will be isolated from a code perspective. Any change to the presentation component need not impact the Business Layer.

Importance of Tiers:

Defining Tiers is important from Security, Transactions and Scalability perspective. Communication between Tiers will draw the attention of Network and Operations to ensure that they are secure and scalable. Also, the deployment strategy of the application will depend on the Tiers and tiers define the system topography. Tiers provide a view of how different systems communicate from a Network perspective.

For instance, the database tier might need additional network security measures to ensure security of the data. Similarly, the web tier might need additional network monitoring as it could be exposed to the internet.

Single Point of Failure



Singe Point of Failure

In System Design and Architecture, Single point of failure (SPOF) finds an important place.

A Single point of failure (SPOF) is a part of a system, which if fails, would stop the entire system from functioning. While designing systems, it is very crucial to identify single point of failures. There could be one or more single point of failures in a system and we need to implement strategies to mitigate them.

For instance, in the above diagram, in Example 1, the ‘centralized coordinator’ is a single point of failure, since if it fails, the entire system would be paralyzed due to lack of coordination. Similarly, in Example 2, the ‘Transformation Service’ is a single point of failure, since if it fails; the entire system would collapse as it is critical to the communication between the service layer and the data service layer.

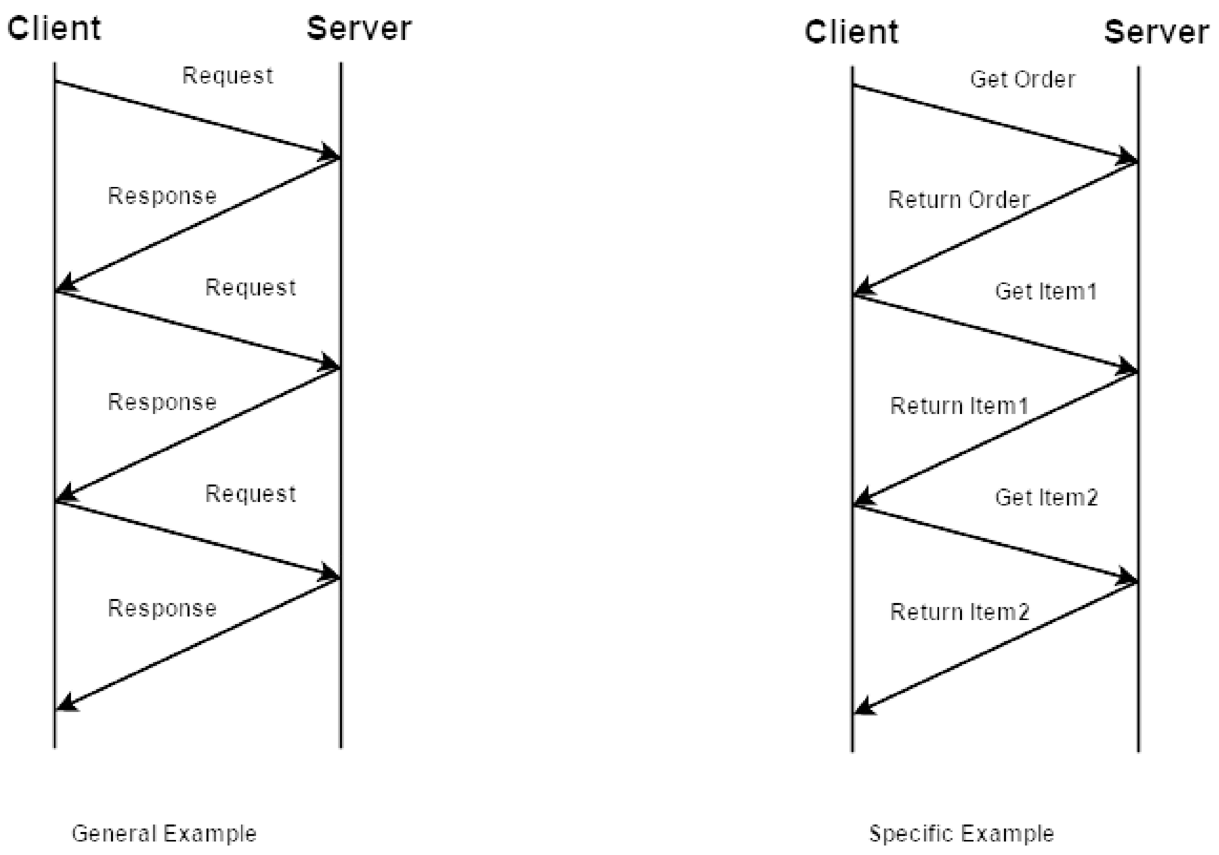
How do we fix single point of failures?

One of the approaches is to add ‘Redundancy’ and use a load balancer. Redundancy is a duplication of critical components with the intention of increasing reliability of the system. For data nodes, we add redundancy through a process known as ‘Replication’. For compute nodes, we add redundancy by adding additional similar compute nodes.

Analysis of single point of failures can happen at various levels in a system as outlined below:

1. Application level SPOF – In this category, different application components are analyzed to find out single point of failures. For e.g., web servers, databases, queues, critical modules like authentication servers etc.
2. Network level or Physical SPOF - In this category, the physical network is analyzed to find out single point of failures. For e.g., Network switches, Routers, Proxy servers, Storage servers etc.

Chatty Applications



Chatty Applications

In this section, we will take a look at Chatty applications, which is a specific characteristic of certain type of applications. There are two ways to explain Chatty Applications as below:

1. Application point of view
2. Network point of view

1. Application point of view

If an application/system consists of sub systems which have to communicate with each other for a number of times, to complete a functional operation, then it is called a ‘Chatty Application’.

For e.g., let us say a client requires order details and its items information from the server. The server might expose an API to return order details and number of items. It could expose a separate API to return item details for a given item number. (Refer above diagram for reference).

So the client has to make one call to get order details and then ‘N’ calls for each of the N items. If the client needs stock information for every item, then it would have to invoke additional N calls to retrieve stock information. So the client would need 2N+1 calls to get an order.

This kind of application could be classified as a ‘Chatty Application’.

Disadvantage of Chatty Applications:

Chatty applications require a lot of network round trip and could cause latency issues.

Advantages of Chatty Applications:

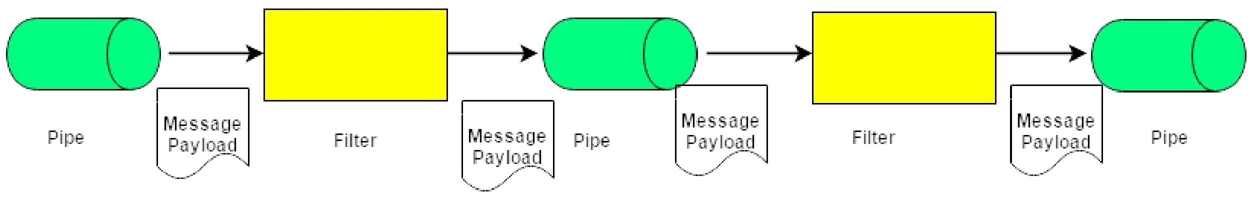
In some cases it might be good to build a chatty application. If we do not build a chatty application, then, systems would need to exchange huge amount of data during the initial request. In the above example, the first request to get order details could return order details and all the associated item details. As a result, the amount of data exchanged per request would be huge. This would also slow down the response to the initial request. We need to analyze the use case and figure out, whether the client really needs all the data at one shot and then chose the right solution. This is basically a tradeoff between response time per request and the network/server load.

2. Network point of view

A chatty protocol is an application or a routing protocol that requires a client or a server to wait for an acknowledgement before it can transmit again. This wait time could be negligent in a Local Area Network, but could get significant in a Wide Area Network and affect system performance.

The solution to this is to use WAN optimizer tools or accelerator tools which will minimize the number of round trips possibly by caching the responses.

Pipes and Filter Architecture



Pipes and Filter Architecture

This is an architectural style or pattern (specifically an Enterprise Integration Pattern), which consists of nodes or components (known as filters) which are connected together by connectors (known as pipes). The filters act on the data i.e. they perform data processing, whereas, pipes transport the data.

Here, pipes are generally dumb and filters are smart. Pipes represent channels which transport data using a particular protocol (for instance, HTTP or FTP etc.). Filters represent processing units which act upon the input data and generate a result, which is fed to the output channel.

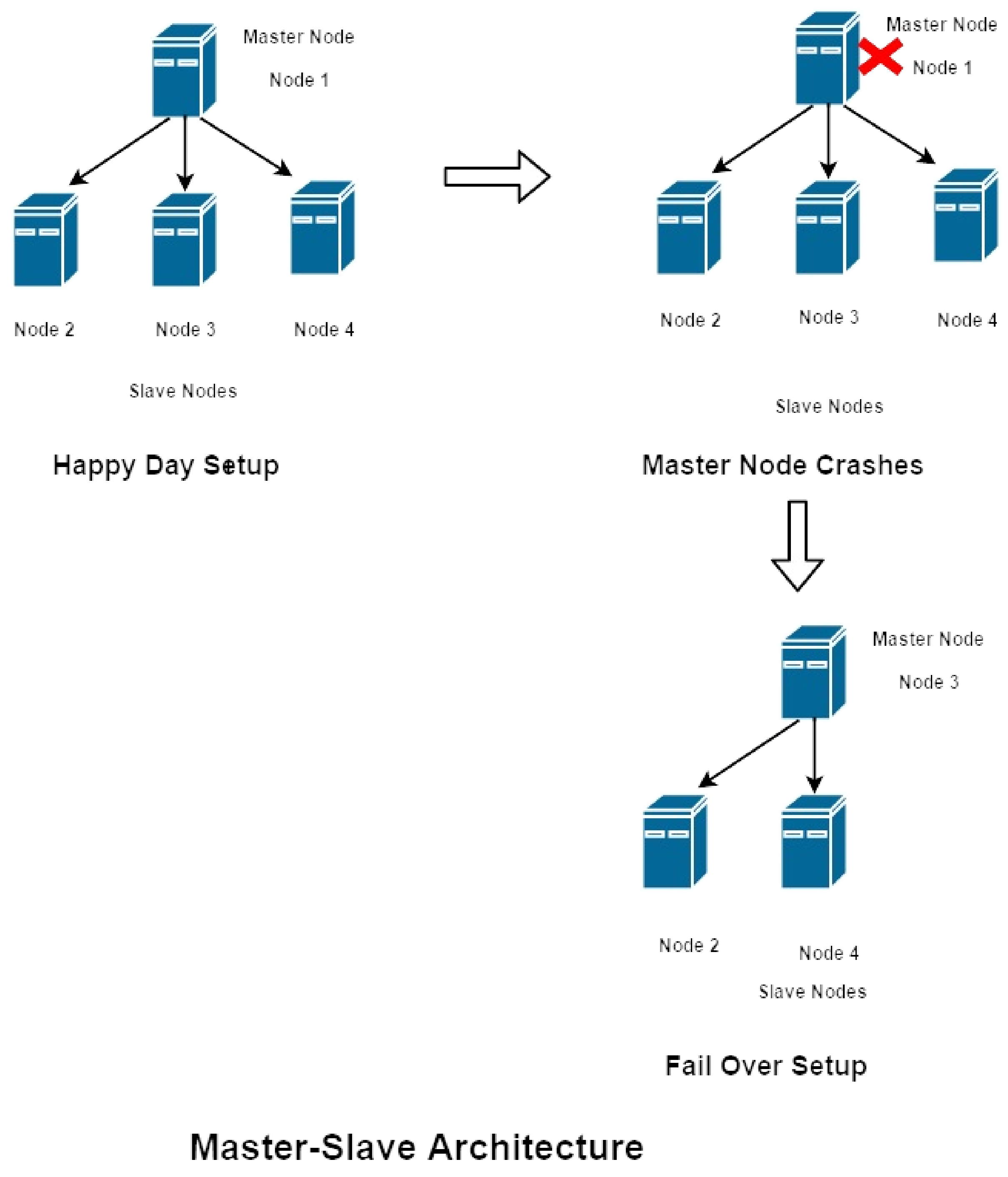
Benefits:

1. Enables loose coupling of systems. If we need a system which has to execute several steps to process data and if these steps need to be configurable, then this architecture suits the best.

Reference Implementation:

Spring Integration module is built upon Pipes and Filter pattern. Similarly, when we have to integrate heterogeneous systems through a workflow, we can apply this pattern.

Master/Slave architecture



In software architecture, Master/Slave defines a type of architecture in which we have one master node and several slave nodes, with the master exerting total control over the slave nodes. Also, the direction of communication in this architecture is always from Master to Slave (However, this may not apply in failure conditions).

The master node could control the following factors (this list is just a sample):

- Communication protocol

- Tasks assigned
- Directions on when to act upon some task

Challenges:

1. Being Fault tolerant:

Let us imagine a scenario in which; the master node fails (refer above diagram). It would have a catastrophic impact on the system, since all the slaves are dependent on the master node for directions.

In such a scenario, we can have an arrangement wherein, when the master node fails, one of the slaves would take over as master without impacting the system. For example, in the above diagram, when the master node crashes, Node 3 is elevated as Master node.

How is the slave node chosen?

- Through configuration. i.e., preconfigure a slave node to act as master
- Through voting. i.e., initiate a voting process amongst the slave nodes to choose the master candidate.

To effectively implement the above fail over strategy, the slave node should be aware of the state of the master before it went down. i.e., it should periodically synchronize the master's state with itself. The time period of synchronization would depend on the use case and the criticality of the system being built.

Reference Implementations:

Databases like MySQL could be configured in a Master- Slave setup. The master node could be read-write node whereas the slave nodes could provide read only operations.

Scalable Architectures

Note: The terms 'machine' and 'node' could be used interchangeably in this section. A machine corresponds to a compute power (with memory and CPU) and need not necessarily mean a physical machine. It could also be a virtual machine.

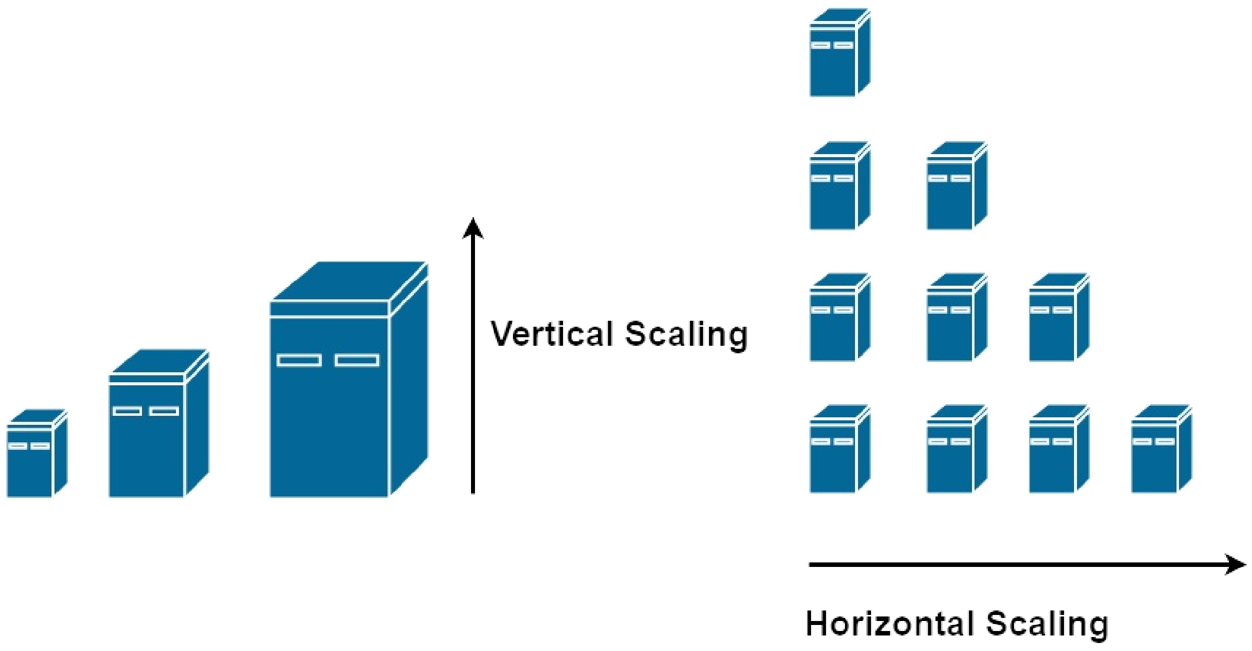
Let us take a look at what is Application scaling and what are the different ways in which we can scale an application.

What is Application Scaling?

It is a process of adding (or removing) resources to an application, so that it can handle a growing (or shrinking) workload.

There are two ways to scale an application:

1. Vertical Scaling
2. Horizontal Scaling



Vertical scaling or 'Scale-In' or 'Scale Up/Down':

It is a type of scaling, where we add (or remove) CPU or memory to servers or machines.

Scale Up example: Increase the CPU from 4 to 8.

Scale Down example: Decrease memory from 1GB to 512MB.

Benefits:

1. Easier to achieve.
2. Requires limited changes to the application (since most modern applications support multi-threading, adding more CPUs should help).

Challenges:

1. There is a limitation on how much we can really scale up, since it depends on the maximum CPU/Memory supported by the hardware.
2. Scaling might require application down time.

Horizontal scaling or 'Scale-Out':

It is a type of scaling, where we add (or remove) machines and connect them together to form a distributed computing system. So instead of one huge super powered machine, we can use a large number of machines with lower configuration (also known as commodity hardware).

Benefits:

1. No limitation on scaling capacity
2. Scaling doesn’t require application down time.
3. Inexpensive, because of lower configuration hardware

Challenges:

Distributed computing introduces its own set of challenges, like Fault tolerance etc. which are covered in a separate section.

Reference Implementations:

Most of the cloud providers today (like AWS, Cloud Foundry, OpenShift etc.) provide the ability to scale out as well as scale in.

Distributed Architecture

In this section, we will take a look at distributed architecture and the challenges it poses.

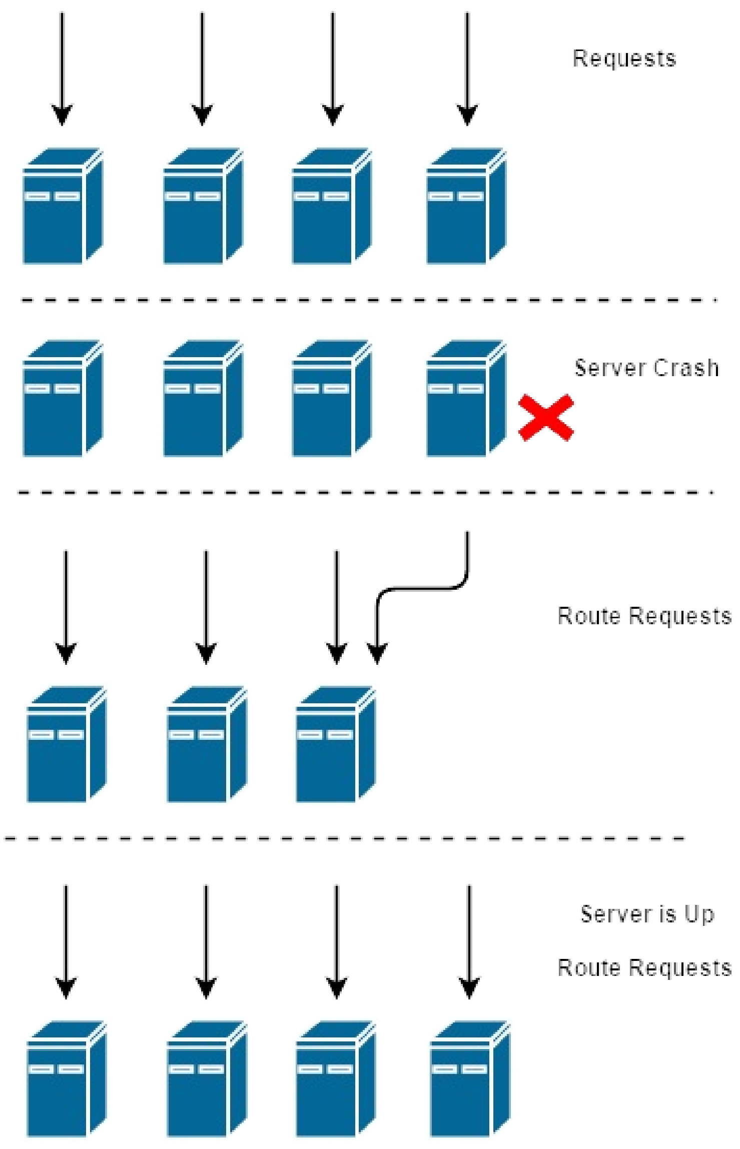
A distributed architecture is an architecture, wherein the resources (either computing workload or data nodes) are spread across several nodes or machines. A distributed system will have redundant nodes (i.e. more than one node which will do the same compute operation, or more than one node which will store the same copy of data). The redundant nodes are to ensure high availability in the case of a node failure.

However, distributed systems are not easy to design and they come with their own set of challenges. Let us take a look at the challenges they pose.

Challenges of Distributed Systems

1. Being Fault tolerant
2. Know and Act
3. Centralized coordination
4. Monitoring and Troubleshooting

1. Being Fault tolerant



Fault Tolerance

In a distributed environment, the system should expect and handle machine failures (the failure frequency will be even higher when commodity hardware is used). There are two aspects of a fault tolerant system:

- If the machine was performing computational workload and it fails, then, another machine should be able to perform the same computation.
- If the machine was storing data and it fails, then, the copy of data should be available in another machine.

The system architecture has to factor in the above two aspects, in order to make the system fault tolerant.

2. Know and Act:

In a distributed environment, when we add (or remove) machines, the system should know about the machines which are added/removed, without manual intervention. Also, the system should act accordingly, to distribute the workload to new machines which were added and stop assigning workload to machines which were removed.

This process, of knowing about addition/removal of machines and acting on it, is critical for a distributed system.

3. Centralized coordination:

A few distributed systems would allow one computation to be split into multiple smaller machines and then collate the results together. These kinds of systems would need a centralized system (or a machine), which would possess the following characteristics:

- Have knowledge of what part of computation is assigned to which machine.
- Ability to give direction to other machines to carry out specific tasks.
- Ability to delegate the task to some other machine if one of the machines fails.
- Ability to collate the results from all other machines to produce a final output.

4. Monitoring and Troubleshooting

In a distributed environment, when we have a large number of machines, monitoring which machine is up/down and tracking the health of each machine becomes very important. This becomes a necessary evil, rather than a sufficient one. Likewise, troubleshooting issues in a distributed environment can get challenging, if the scope of the work being analyzed spans multiple machines.

Reference Implementations:

Cloud solutions like Kubernetes, platforms like Amazon Web Services etc. offer an ecosystem to build distributed systems. Frameworks like Apache Zookeeper provide a lot of features for coordinating distributed systems.

Fault Handling

It is common for systems to fail. Failures could be network failures, application logic failures, storage failure, infrastructure failures etc.

It is important for systems to handle failures gracefully and make the system robust enough.

There are two error handling patterns which are commonly used in system design.

1. Fail safe
2. Fail fast

Let us take a look at each of these:

1. Fail safe:

In this pattern, the system is designed to fail in such a way that it doesn't affect the overall system behavior and doesn't disrupt normal functioning. This might temporarily hide the problem, however, it would resurface as a bigger problem later on, and at that stage it could display different symptoms and will be generally harder to trace and figure out the root cause.

2. Fail fast:

In this pattern, the system is designed to fail in such a way that it fails quickly if there is a fault and is allowed to disrupt the normal functioning of the application. This allows the problem to be identified and addressed quickly. The downside of this approach is that, it would disrupt users and normalcy. However, the short pain inconvenience would prove hugely beneficial and cost effective, since the problem is tackled upright.

Retry mechanisms in fault handling:

While dealing with errors, retry mechanisms are very popular and should be considered when we have multiple system integrations. Let us say, system A (or service A), invokes system B (or service B). If system B is down or throws an error, then system A can implement a mechanism, wherein system A would retry the same invocation, with the same request data periodically, until system B responds successfully. This mechanism would make sure that the error transactions are not lost in that time window (when system B was down).

This kind of implementation would require system A to persist the request data, so that it could be used during retry.

Infrastructure

This section explains the various Infrastructure Architecture concepts in detail.

IP Addresses

In this section, we will take a look at IP Address ranges and related concepts as they are essential to understand a little bit of Infrastructure Architecture.

IP address is used to identify machines in a network. IP address could be either 32 bit (IPv4) or 128 bit (IPv6).

IPV4:

Let us discuss IPv4 in more detail.

IP addresses are generally expressed in decimal format, for e.g., 192.128.160.164. The corresponding value in binary format is obtained by using binary representation of each of the individual numbers using 8 bit representation as shown in the below table.

Decimal Representation	Binary (8 bit) Representation
192	11000000
128	10000000
160	10100000
164	10100100

An IP address has two parts to it. The network address and the host machine address. To determine which part is IP address and which part is host machine address, we need another representation called 'Subnet Mask'.

Subnet Mask:

The subnet mask is also represented in a similar way as IP addresses. For e.g., a sample subnet mask could be 255.255.255.0

When we perform a logical AND operation between the IP address and the subnet mask, we get the network address. For e.g., IP address: 192.128.160.164

Subnet Mask: 255.255.255.0

Logical AND result (Network Address): 192.128.160

In TCP/IP Protocol, a router will use the subnet mask to determine the network address and determine the network in which a machine exists.

Network Mask classes:

Extending on the above concept of subnet mask, there are 3 classes of Network masks as defined below. These three classes have been defined and they have been allocated IP addresses for each of these classes by an International Organization known as InterNIC.

Class	Mask	Starting IP Address Range	# of Host machines
Class A	255.0.0.0	10.0.0.0	16,777,216
Class B	255.255.0.0	172.16.0.0	1,048, 576
Class C	255.255.255.0	192.168.0.0	65,536

A Class A network can accommodate maximum number of hosts, followed by Class B and Class C.

CIDR Notation:

Let’s say in Class C network, we take most significant 3 bits out of the last decimal notation for the subnet mask. So the last decimal representation would be 11100000.

So the subnet mask would become 255.255.255.224

The total number of bits used for subnet mask, thus becomes (8 X 3) +3 = 27

We can represent this subnet mask as /27. This representation is known as CIDR notation.

Public and Private IP Addresses:

A public IP address is the address that is assigned to a computing device to allow direct access over the Internet. Public IP addresses should be unique globally as it is used for routing traffic requests over the internet. Web Servers, Email Servers etc. generally would have a public IP address.

A private IP address is the address space allocated by InterNIC to allow organizations to create their own private network. These private IP addresses could belong to any one of the class A, B, C networks defined above.

Network Address Translation:

Consider a Local Area Network within a corporation, which has a huge number of workstation machines. These machines need access to internet; however, the corporation cannot assign a public IP address to each of these machines (For reasons of security and economy). All these machines would have a private IP address. But in order to connect to the Internet, and be able to send/receive traffic, these machines would need some kind of a public IP address.

Network Address Translation is a concept which will help us achieve this. In a LAN, the Network Address Translation is performed by a Firewall, which will have a public IP Address. When a workstation wants to send a request to a resource on the internet, the traffic goes via the Router, and the Router would recognize that this request is intended to be routed to internet and will forward it to the firewall. The resources outside the firewall see the request as coming from a public IP address.

Thus, the requests coming out from all the workstations will have the same Public IP Address.

Mail Server Protocols

In this section, we will take a look at the protocols used in an Email server.

1. SMTP: Simple Mail Transfer protocol
2. POP3: Post Office Protocol
3. IMAP: Internet Message Access Protocol

1. SMTP (Simple Mail Transfer protocol):

This protocol communicates through port 25. This protocol is used when email is delivered from email client to server. E.g. Outgoing emails from outlook server to an external email server.

2. POP3 (Post Office Protocol):

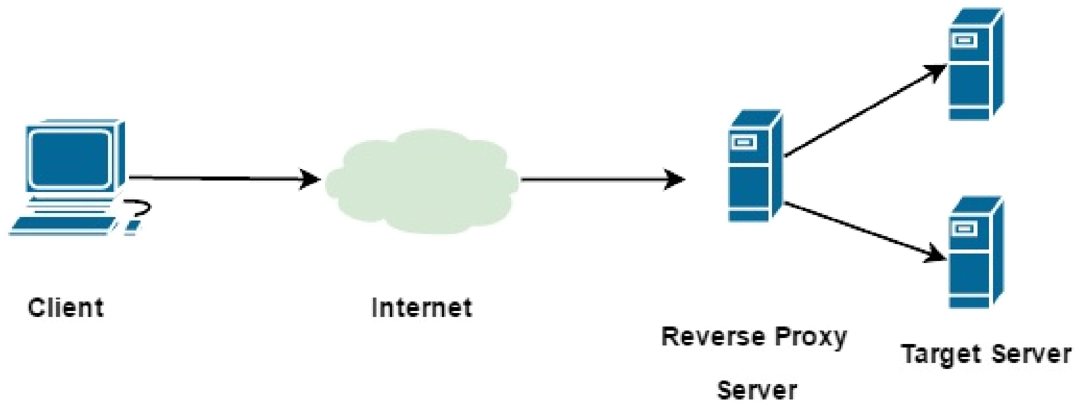
This protocol communicates through port #110. This allows email client to download email from the server. This protocol downloads all the emails and then deletes the emails from the server. E.g. A desktop mail client can use POP3 protocol.

3. IMAP (Internet Message Access Protocol):

This protocol allows email clients to download emails from the server. Unlike POP3, it doesn’t delete the email from the server. If IMAP is used, more space needs to be provisioned on the server.

IMAP is used when there is a need to access the email from a variety of clients like desktop, mobile, tablet etc. Since the emails are always stored on the server, unless deleted explicitly, it could be accessed from different clients.

Reverse Proxy



Reverse Proxy

A reverse proxy is a type of a proxy server which sits in front of the actual server (to which the requests are intended to be sent). A Reverse proxy accepts requests from the client and forwards the requests to the actual server. Typically, the reverse proxy server resides in a DMZ. The client would never know that it is sending requests to a reverse proxy server (whereas in the case of a proxy server pattern, the client is cognizant of the fact that it is communicating with a Proxy and not the actual server).

Why do we need a Reverse Proxy?

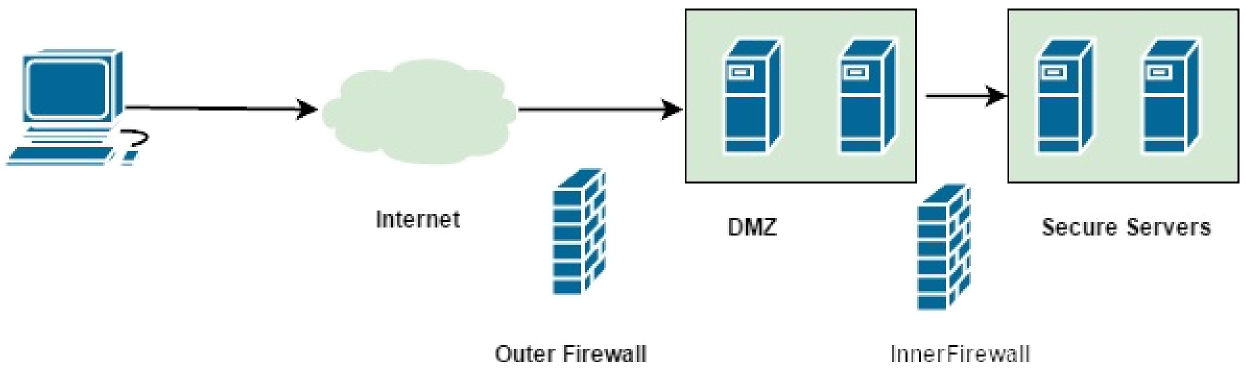
A Reverse proxy is generally used for a variety of purposes as mentioned below:

1. Load balancing
2. To implement Gateways
3. To implement security
4. To implement caching

Reference Implementations:

Apache server, Nginx server or a Custom implementation.

DMZ (Demilitarized Zone)



DMZ

DMZ in general refers to an area which lies between the borders of two countries. It would be generally agreed upon by both the countries, through treaties, to not claim ownership of that area.

In the software world, DMZ refers to the part of the network between the internet (which is an untrusted network) and the corporate network (which is a trusted network). The servers or resources in the DMZ are not as secure as those in the LAN (Local Area Network), but not as insecure as those in the internet. (Refer above diagram).

The resources in the DMZ are exposed to the internet as they will have to interact with the web. However they also need to interact with the resources in the LAN, so the firewall in between them restricts access. Generally, there will be a firewall between the DMZ and the internet and a different firewall with stricter access rules between the DMZ and the intranet.

What resides in DMZ?

Some examples of servers which reside in the DMZ are provided below:

1. Mail Server
2. FTP Server
3. Web Server

Why do we need a DMZ?

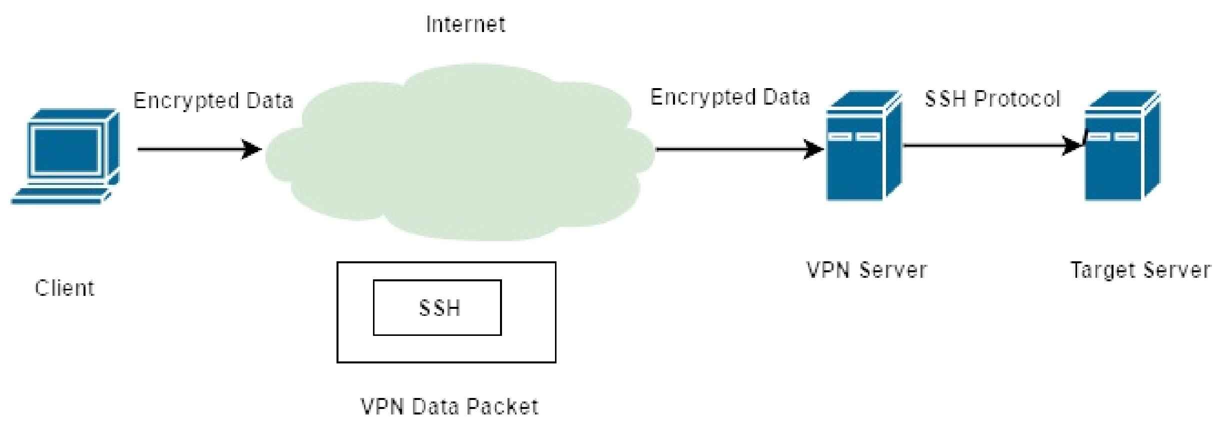
Having a DMZ ensures that we do not expose all our resources to the internet. If we did not use a DMZ, then it would be difficult to secure all these resources, and the entire corporate network would be at risk.

So we expose only a small part of the resources to the web by placing them in the DMZ. This reduces the overall surface area of attack. The resources in the DMZ act as first line of defense against attacks from the internet. The resources in the DMZ could be closely monitored and the firewall between the DMZ resources and the LAN provides adequate security. Also, the resources in DMZ cannot initiate requests, they can only forward requests.

DMZ in Application Architecture:

From an application architecture point of view, in internet facing applications, the load balancers are generally placed in the DMZ, since they receive requests from the web. The load balancers then interact with the servers behind the firewall. The servers like database servers etc. which should be protected, would lie behind the firewall within the corporate network.

Tunneling



Tunnelling

In Network security, we often come across a term known as ‘Tunneling’, which is explained below.

Tunneling (also known as ‘Port Forwarding’) is a process in which data is sent using unsupported protocol over the network through encapsulation. For instance, let us say we want send data using protocol X, whereas the network supports protocol Y. To accommodate protocol X, we can encapsulate the protocol X data packets and send them over the network using Protocol Y. At the receiving end, the server would retrieve the data packets and decode the data to retrieve the protocol X data packets.

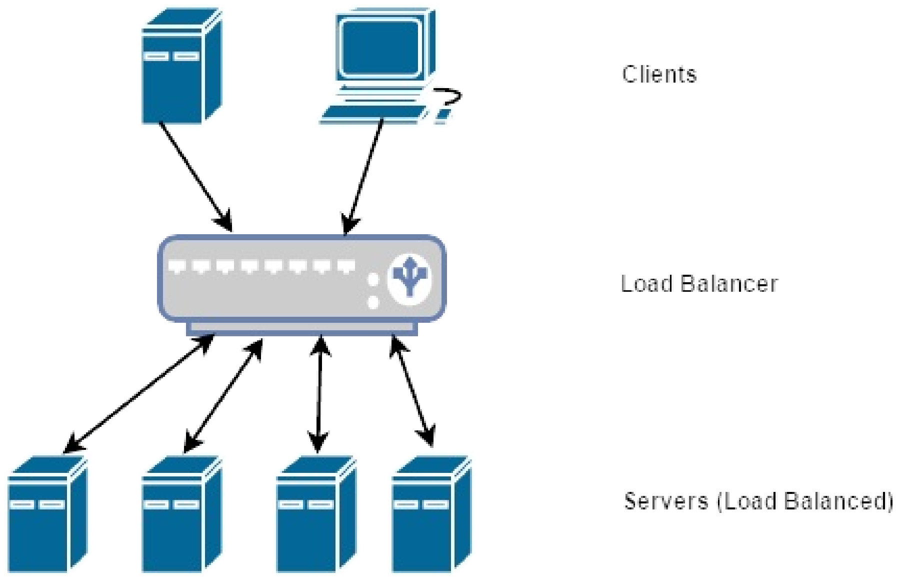
The tunneling protocol works by using the data portion of a packet (the payload) to carry the packets that actually provide the service. Tunneling uses a layered protocol model such as those of the OSI or TCP/IP protocol suite, but usually violates the layering when using the payload. The payload is made to carry a service not normally provided by the network. Typically, the delivery protocol operates at an equal or higher level in the layered model than the payload protocol.

As a practical illustration, the most common form of tunneling is Virtual Private Networks (VPN). Let us say, we want to connect to a corporate network through VPN and try to get access to a server through SSH protocol in the corporate network. The SSH protocol data packets get encapsulated in the network layer protocol and are sent to the VPN server. The VPN server would decode the data packets, decrypt the message, retrieve the actual data packets and route it to the actual server through SSH protocol. The above diagram illustrates this process.

Using Tunneling to get across firewalls:

We can use tunneling to pass requests through firewalls and snoop into the servers. This could be done by using protocols which are generally blocked by the firewalls, by wrapping the requests into say Http protocol which are not blocked by the firewalls.

Load Balancer



Load Balancer

A Load Balancer distributes traffic or incoming requests across multiple resources (i.e., data nodes or compute nodes).

Why do we need a Load Balancer?

Let us say we have X nodes (or machines), which either store same data (i.e. Replicated data) or perform same computation. We can keep adding additional similar nodes. Now, when a request comes in (either to perform data operation or to perform computation), we need to route this request to one of these nodes.

A Load Balancer performs the above mentioned job of routing requests. It prevents a scenario where in, one single resource or node is overloaded with excess work, whereas the other nodes are idle. A Load Balancer also assists in failover implementation, as they will not route requests to failed nodes. A Load Balancer maintains a registry of all the nodes and it would send heartbeats to each of the nodes regularly, to check if a node is alive and will update its registry accordingly. It will also dynamically update its registry if a node is added or removed.

Routing criteria:

A Load Balancer generally routes requests based on one of the below criteria:

1. Round robin – In this mode, the nodes are selected one after another in a round robin basis.
2. Priority – In this mode, some nodes are given preference over other.
3. Least connections – In this mode, the request would be routed to the node having least number of connections.

Load Balancer Types:

Load Balancers could be categorized into two categories and each of the categories could be further divided into different types as below.

Category I:

1. Hardware Load Balancer
2. Software Load Balancer

Category II:

1. Server side Load Balancer
2. Client side Load Balancer

Category I:

1. Hardware Load Balancer:

Hardware Load Balancer is a preconfigured machine with processor, memory, Operating System etc., optimized for high performance and the Load Balancer software installed on it.

Reference Implementation:

F5 is a well-known hardware Load balancer.

2. Software Load Balancer:

Software Load Balancer comprises of just the Load Balancer software and the buyer would have to install and configure it on their own. When compared to Hardware Load Balancer, the right memory and Processor configurations would have to be configured for optimal performance.

Reference Implementation:

HAProxy is a well-known software Load balancer

Category II:

1. Server side Load Balancer:

In this type of load balancing, the Load balancing and routing happens on server side. We can use more than one server side Load Balancer either in a master/slave or a clustered mode. The client need not even know that it is hitting a Load Balancer. Load Balancing would be transparent to the client and the load balancer acts a reverse proxy.

Reference Implementation:

Consul is a server side Load balancer

2. Client side Load Balancer:

In this type of load balancing, the Load balancing and routing happens on client side. The client should be aware of all the nodes and their addresses. The client uses an algorithm to select a suitable node to route the request.

Reference Implementation:

Eureka is a client side Load balancer

Client Side Vs Server Side Load Balancing:

Advantages of Client Side over Server Side:

Since load balancing happens on client side, there is no single point of failure, whereas addition of a huge number of clients might burden the server side load balancer.

Disadvantages of Client Side over Server Side:

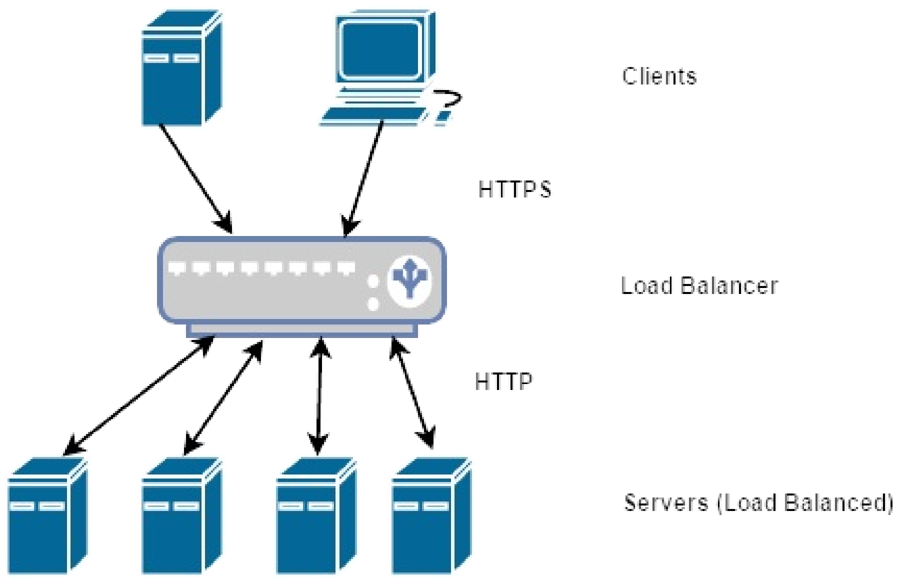
1. Each client will be aware of the load distribution on the nodes originating from that client alone and it will not know the overall load on any node. So, the overall load distribution on the nodes might not be even and could be biased.
2. The client needs to know about all the nodes and when any node is added or removed, all the clients need to be notified.
3. Additional burden on client to perform load balancing.

Despite of its disadvantages, Client side load balancers are hugely popular in microservices based cloud native applications, where huge number of clients keep getting added as consumers for a service, and server side load balancing might introduce single point of failure.

SSL Termination and Pass Through

In Infrastructure world, we often come across terms like SSL Termination and SSL Pass through. We will take a look at each of these in the below section.

SSL Termination:

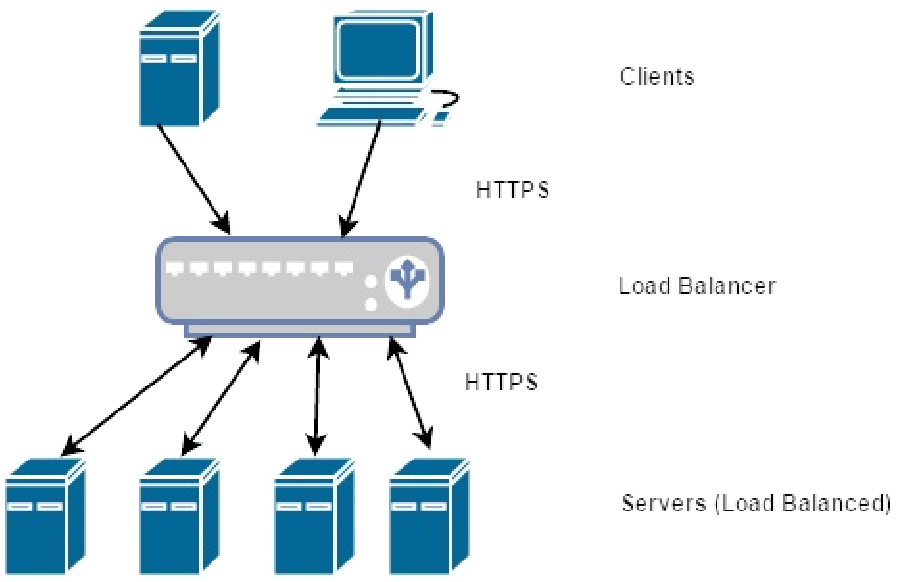


SSL Termination

SSL Termination is a process wherein the encrypted secure traffic (HTTPS) is consumed by a server, performs decryption and then the server forwards unencrypted request (HTTP) to other servers in the network. The other servers in the network are assumed to be in a secure network and they will not be exposed to the internet.

Generally, a load balancer performs the task of SSL Termination.

SSL Pass Through:



SSL Pass Through

SSL Pass through, in contrary to termination, is a process wherein the encrypted secure traffic (HTTPS) is forwarded as is by the server to other servers in the network.

Load balancers could be configured to use SSL Pass through. In this scenario, the individual servers should be capable of decrypting the SSL request.

SSL Termination vs SSL Pass through:

1. When SSL pass through is used, the SSL certificates need to be maintained in individual servers whereas in SSL termination, the certificates need to be maintained in a centralized server (Load Balancer). Maintaining certificates in a centralized place is generally easy to manage. Also, it is easier to apply security patches and easy to manage SSL security at a central place as opposed to performing that in a lot of different servers.
2. SSL Pass through is more secure since the traffic is secured end to end, whereas in SSL termination, the insecure traffic after termination could be vulnerable.
3. In SSL Termination, it is easier to handle DDOS (Distributed denial of service) attacks over SSL as most modern load balancers provide safety against these attacks.

Reference Implementation:

Servers capable of performing SSL Termination or Pass through:

1. Apache HTTP Server
2. HAProxy

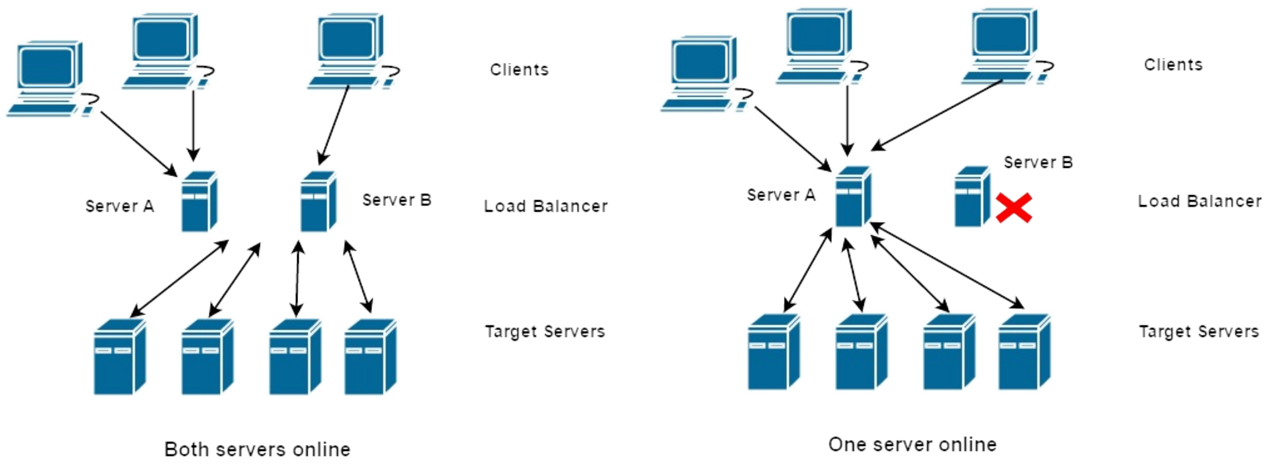
3. Nginx

Active-Active Active-Passive configurations

Generally in cluster configurations and Load balancer Configurations, we come across the terms Active-Active and Active-Passive configurations. We will take a look at these configurations in more detail in this section.

Note that for simplicity, we have used two servers for explanation. The count is only for illustration purpose and could be higher.

Active-Active Configuration:



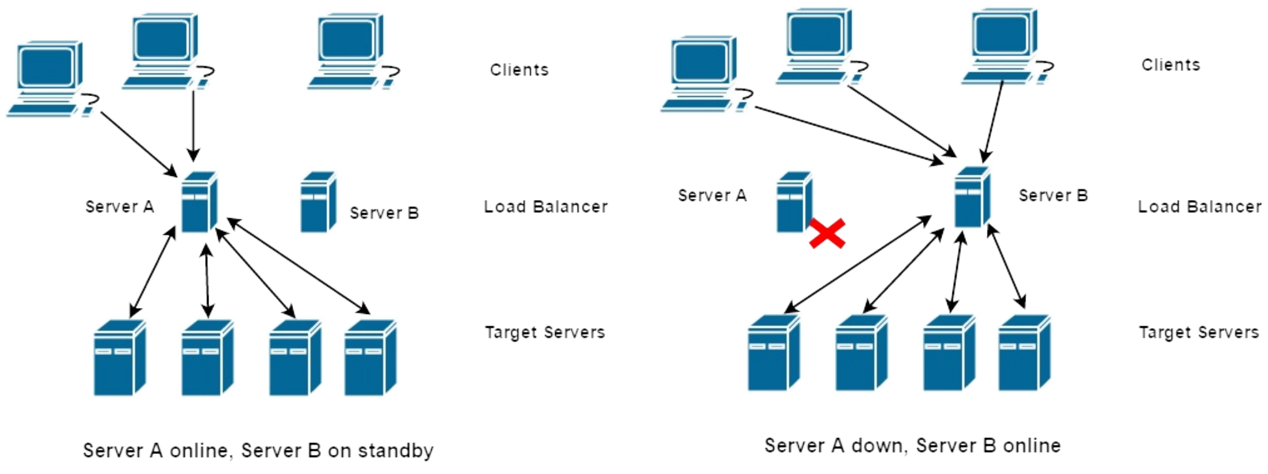
Active Active Configuration
(Eg Load balancer)

In this configuration, two servers would be online together and would independently share the workload and work as peers.

Fail over:

When one server goes down, the other server would cater to all the requests. In the above diagram, when Server B fails, server A would cater to the requests.

Active-Passive Configuration:



Active Passive Configuration
(Eg Load balancer)

In this configuration, only one server would be online at any given time and the other server would be only configured as a fail over server and would remain offline (or passive). When the primary server (Server A, in the above diagram) goes down, the second server would come online and start catering to the requests.

Active-Active Vs Active-Passive configurations:

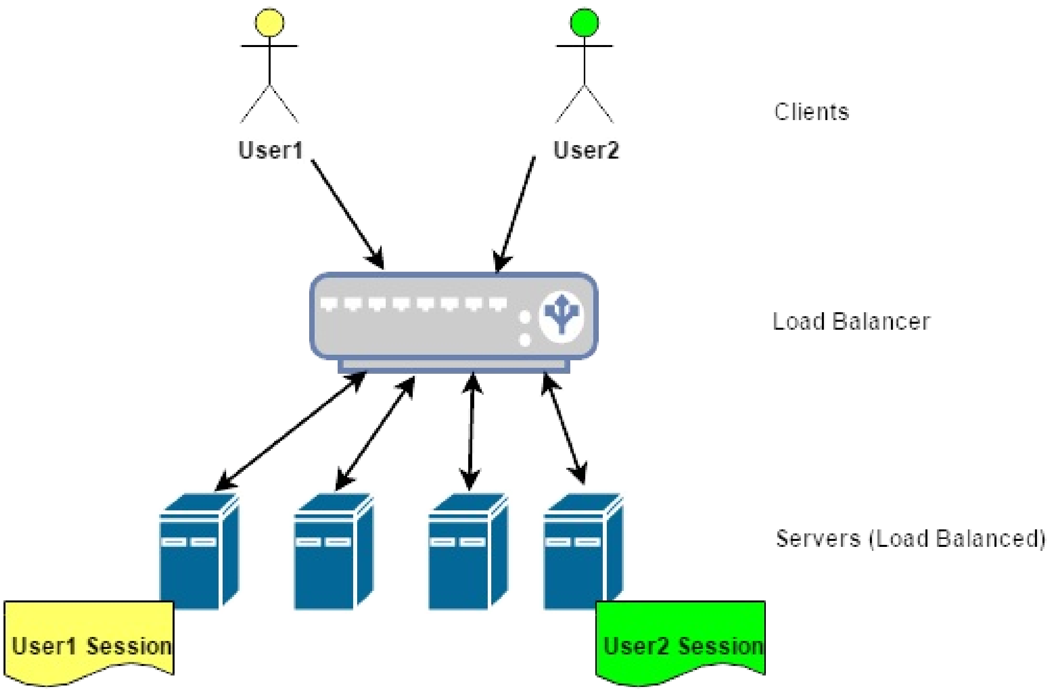
Active-Active	Active-Passive
# Of servers online in a happy day scenario < # of servers online in a fail over scenario.	# Of servers online in a happy day scenario = # of servers online in a fail over scenario.
System performance suffers during fail over	System performance remains same during fail over

Reference Implementations:

Load balancers could be configured to work either in Active-Active or in Active-Passive configurations. Similarly, few relational databases support either of these configurations. Active-Active or Active-Passive configuration could be used as a general design principle in System design.

Sticky Sessions and Session Replication

Sticky Sessions:



Sticky Session

Traditionally, web servers used to store HTTP sessions in web applications. These sessions would store user data for a specified time frame (which is known as session time out).

Load balancers generally support a feature known as ‘Sticky Sessions’ (Also known as ‘Server Affinity’). If this feature is turned on, the load balancer remembers which server was assigned to which session (based on session Id or client’s IP address). The load balancer would route all the future requests within a time frame to the same server.

The time till which a load balancer remembers (or persists) this information should be ideally more than the server’s session time out.

If a load balancer server goes down, the users connected to it lose their session.

Relevance of Sticky sessions in Cloud Architecture:

Sticky sessions are not preferred in cloud based architecture as we will have a huge number of application servers which are spawned and torn regularly. Maintaining sticky sessions would defeat the purpose of scaling, as requests might be unevenly balanced since the requests would hit the same set of servers based on server affinity.

Session Replication:

Instead of sticky sessions, a concept known as ‘Session Replication’ is preferred in cloud based systems. In Session Replication, the HTTP session is not stored along with the application, but is generally stored in a distributed cache (like for instance Redis). Since the session is stored in a distributed server, we can scale the application horizontally without worrying about maintaining the session state.

Servers

In this section, we will take a look at different types of servers. There are three types of physical Servers:

1. Tower
2. Rack
3. Blade

1. Tower Servers:

This type of server looks like a desktop cabinet. This type of server does not require a great deal of maintenance and is useful for small companies who want to begin using servers for relatively low end jobs. Generally, a tower server can accommodate up to 6 drives and two processors. For a large number of tower servers, cabling can get clumsier and would also consume a lot of space.

2. Rack Servers:

A rack server is designed to be positioned in a bay, which enables you to stack various devices on top of each other in a large tower. The bay will accommodate all of the hardware devices the company needs to function, including the server, storage devices, and security and network appliances.

The benefit of this type of server is that, having all components of the system located in the same place (in a single Rack) makes it easier to manage connections and maintain the system. The bay is, in a way, the 'data center' of the organization.

3. Blade Server:

Blade server is the latest development in the history of the different types of servers. Slim and compact, just like a blade, they slide vertically into a specially designed chassis. They share certain elements of the hardware with other blade servers in the chassis for the purposes of efficiency and cost reduction. Blade servers, for example, use a single feed positioned on the host compartment.

Blade servers will give us much greater processing power, take up less space and use less energy than other forms of servers used for the same purposes. However, Blade servers are expensive when compared to Tower or Rack servers.

Storage Devices and Networks

In this section, we will take a look at the various types of storage devices and storage networks. Storage plays an important role in any system and having a high level understanding of this could be essential.

Storage Devices:

This topic would explain the different types of storage devices which are presently in use.

Broadly there are 6 types of storage devices:

1. Solid State Drive
2. Flash
3. All Flash Array
4. Block based storage
5. Object Storage

1. Solid State Drive:

A Solid State Drive is a storage device that uses integrated circuit assemblies as memory to store data persistently. SSD technology primarily uses electronic interfaces compatible with traditional block input/output (I/O) hard disk drives. SSDs have no moving mechanical components. This distinguishes them from traditional electromechanical magnetic disks such as hard disk drives (HDDs), which contain spinning disks and movable read/write heads. SSDs have lower access time, and lower latency.

2. Flash Storage:

Flash storage is a non-volatile computer memory with an integrated circuit that does not need continuous power to retain the data, but is a bit more expensive than magnetic storage. Modern SSD hard drives are Flash-based, so today there’s not really a difference today between SSD and Flash. SSD is simply a disk that doesn’t have moving parts, and Flash is the implementation that allows that to happen.

3. All Flash Array:

An all-flash array is a solid state storage disk system that contains multiple flash memory drives instead of spinning hard disk drives.

4. Block Storage:

Block storage is a type of data storage where data is stored in volumes, also referred to as blocks. Each block acts as an individual hard drive. These blocks are controlled by the server-based operating system, and are generally accessed by Fibre Channel (FC), Fibre Channel over Ethernet (FCoE) or iSCSI protocols. While block storage devices tend to be more complex and expensive than file storage, they also tend to be more flexible and provide better performance.

5. Object Storage:

In Object Storage, an object is defined as data (typically a file) along with all its metadata, all bundled up as an object. This object is given an ID that is typically calculated from the content of that object (both file and metadata) itself. An object is always retrieved by an application by presenting the object ID to object storage. Unlike files and file systems, objects are stored in a flat structure.

Storage Networks:

Storage Area Network:

A storage area network (SAN) is a network which provides access to consolidated, block level data storage. SANs are primarily used to enhance storage devices, such as disk arrays accessible to servers so that the devices appear to the operating system as locally attached devices.

A SAN typically has its own network of storage devices that are generally not accessible through the local area network (LAN) by other devices. A SAN does not provide file abstraction, only block-level operations. However, file systems built on top of SANs do provide file-level access, and are known as shared-disk file systems.

Network-attached storage (NAS):

NAS was designed before the advent of SAN, as a solution to the limitations of the traditionally used direct-attached storage (DAS), in which individual storage devices such as disk drives are connected directly to each individual computer and not shared.

SAN vs NAS:

Following are the differences between a SAN and a NAS:

- In a NAS solution the storage devices are directly connected to a "NAS-Server" that makes the storage available at a file-level to the other computers across the LAN. In a SAN solution the storage is made available via a server or other dedicated piece of hardware at a lower "block-level", leaving file system concerns to the "client" side.

- SAN uses protocols like ISCSI, whereas NAS uses protocol like NFS

iSCSI (Internet Small Computer System Interface):

iSCSI is nothing but an IP based standard for interconnecting storage arrays and hosts. It is used to carry SCSI traffic over IP networks. It’s a networking protocol which is built on top of TCP/IP.

RAID:

NAS systems are networked appliances which contain one or more storage drives, often arranged into logical, redundant storage containers or RAID.

RAID (originally redundant array of inexpensive disks, now commonly redundant array of independent disks) is a data storage virtualization technology that combines multiple physical disk drive components into a single logical unit for the purposes of data redundancy, performance improvement, or both.

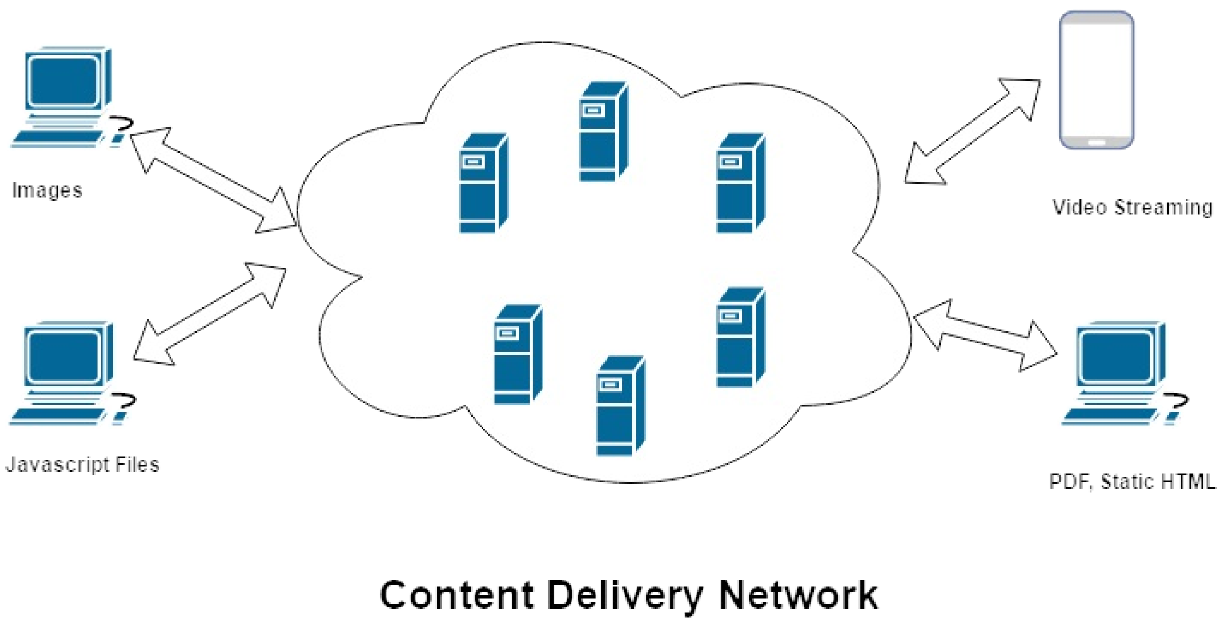
Data is distributed across the drives in one of several ways, referred to as RAID levels (RAID 0 to RAID 6), depending on the required level of redundancy and performance.

How do RDBMS store the data?

RDBMS typically use B+ trees. B+ tree is a special data structure allowing to efficiently store (i.e. access and update) a large sorted dictionary on a block storage device (i.e. HDD or SSD).

Sorted dictionary is, essentially, a phone book: it allows locating a random entry by doing a tiny number of steps - i.e. without reading the whole book.

Content Delivery Network (CDN)



A content delivery network or content distribution network (CDN) is a globally distributed network of proxy servers deployed in multiple data centers, which render media content like images, videos, downloadable files like javascript files, static html files etc.

CDNs provide high performance and high availability. A lot of companies provide CDN as a service offering, where different clients can host their content and the client companies pay for utilizing these CDN services.

CDNs also implement load balancing, caching and compression of media files to achieve high performance.

Importance of CDN in Architecture and Cloud:

In any application, one of the factors which will affect the performance of the User Interface layer is the speed at which the content is delivered to the client. If the static content like javascript files, images, videos etc., is served up from the application server, then it would burden the load on the application server and will cause performance issues. So it is a good practice to use a CDN and serve up media content from CDNs, as CDNs will guarantee high performance.

Reference Implementations:

Amazon Web Services (AWS) provides a CDN ‘Amazon CloudFront’. Similarly, Google provides a CDN for serving Angular JS files.

Latency

Latency is the time delay for a message to reach the intended recipient. Latency could be caused by a variety of factors like hardware, software or by network. Latency affects system performance and is an important criterion in measuring or estimating performance.

Types of Latency: There are different types of latency as follows:

1. Hardware Latency
2. Software Latency
3. Network Latency

Let us take a look at each of the following:

1. Hardware Latency:

Hardware latency is generally induced by the hardware components. For example, the storage devices used like solid state, disk or tapes induce latency.

2. Software Latency:

Software latency is induced by the software components as the name implies. For example, if we are using third party software for load balancing, the software might inadvertently induce latency.

3. Network latency:

If the application components are deployed in geographically distributed regions, then the network latency between the regions should be taken into consideration.

For example, if a database is deployed in American region and the application which accesses the database is deployed in Singapore region, there could be a latency induced by the network. This should be factored in while estimating the overall application response time.

Overcoming Latency:

There are a few ways to overcome network latency. Generally, use of geographically distributed CDNs help in reducing latency, as the resources are served from local CDNs. Also, use of co-located servers help in reducing latency.

Disaster Recovery

In IT infrastructure, Disaster Recovery (known as DR in short), is a process which is put in place to quickly recover to normal business in the event of a failure or disaster.

Tier 1 application:

Applications which require a very high up time and whose failure might cause huge impact on an Enterprise (either financially or in terms of reputation or compliance regulations) are grouped under 'Tier 1 applications'. For instance, banking applications, stock market applications, and air traffic controller applications cannot afford to have a downtime and they need to be up and running all the time.

Disaster:

An infrastructure failure could be because of hardware failure, disk crashes or even natural catastrophes like earthquake, flood or fire etc. Disaster Recovery process typically involves all the steps necessary in order to bring the business back to normalcy.

Recovery:

A Disaster Recovery setup requires the infrastructure to be replicated and redundant setup to be in place, which duplicates the entire infrastructure setup. Also, generally, the DR infrastructure should be deployed in a different geographical region, in order to guard against natural calamities.

A DR setup also requires the data to be replicated across redundant deployments, so that when one infrastructure goes down, the other one can start serving requests immediately. For that to happen, the data persisted by the system (databases/object stores etc.), should be replicated in real time to the redundant system.

Replication:

Replication is a process in which data is continuously and incrementally copied over to another redundant data storage system. This happens in real time. Only modified data is copied over and thus ensures another copy is available in case the primary copy goes down.

Replication types:

There are two ways in which the data could be replicated across redundant systems:

1. Hardware Replication
2. Software Replication

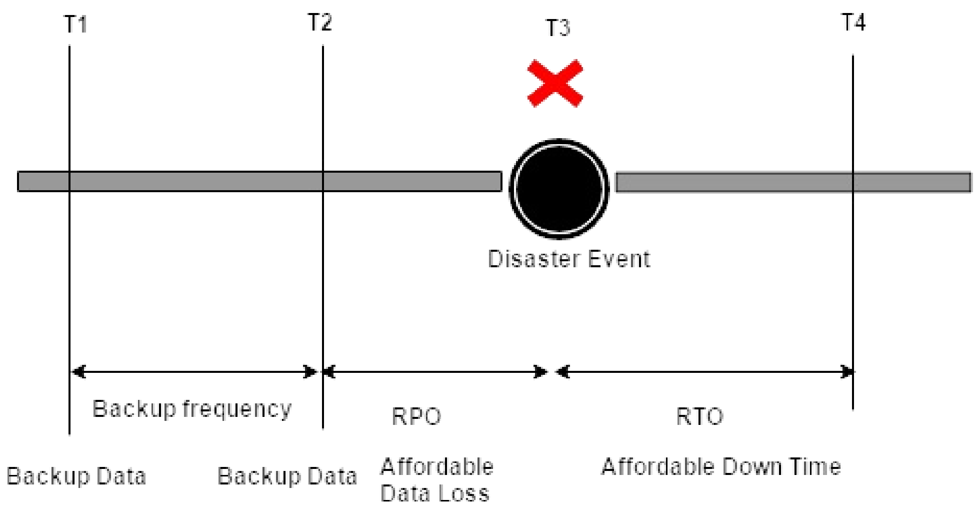
1. Hardware Replication

In this type of replication, the data copy is performed by the storage hardware. The replication happens at the hardware level. One of the disadvantages of this process is that it requires matching hardware at both the sides. It might sometimes become expensive to maintain matching hardware at primary production and backup sites, since generally high performance hardware is used at production and comparatively less performance hardware could be used at backup site. However hardware replication doesn't put too much of an overhead on the production servers as it happens at hardware level.

2. Software Replication

In this type of replication, the data copy is performed by software at hypervisor level or often by database replication software. This type of replication has the advantage that the software is aware of the application state while performing replication and works well with distributed systems as well. It is not a mere block to block data copy as in hardware replication. Software based replication could add additional overhead on performance.

RPO and RTO



RPO and RTO

In Disaster Recovery, there are two concepts named RTO (Recovery Time Objective) and RPO (Recovery Point Object) which are helpful in defining our response to a disaster and they give a measure of how quickly and how much can we recover. Both of the parameters are defined based on the business needs and criticality of the system.

RPO:

RPO is the amount of data loss expressed in terms of time, which a business can afford to lose when there is a disaster. Thus, RPO also indirectly specifies the minimum backup frequency. For example, in the above diagram, the backup is taken frequently at time points T1, T2 and disaster strikes at T3. Obviously, the data generated by the system between T2 and T3 is lost and that difference in time period (T3-T2) is the RPO.

If a backup is taken at 8 PM every day and a disaster happens the following day at 6 Am, then the data between 8 PM and 6 AM is lost. So RPO = 10 hours. Higher RPO indicates lower data recovery and lower RPO indicates higher data recovery.

Note that RPO is not defined by the backup frequency of the system, but it is other way round. i.e., based on the RPO defined by the business needs, the backup frequency is determined.

RTO:

RTO is the time period within which a system should recover and get back to normal running state, after a disaster has struck. For example, in the above diagram, if the system is expected to come back online at T4, then RTO is the down time period (T4-T3).

If RTO= 0, then it means, the system cannot afford to go down. The system should be up and running 24X7X365. Lower RTO indicates higher availability.

Data Architecture

This section explains the various Data Architecture concepts in detail.

Database Index

In this section, we will take a look at Database index, types of index and how it works.

An index is used to assist in faster data retrieval. An index is generally created after analyzing retrieval patterns. A primary key is by default an indexed column. We can also create custom indices based on one or more columns.

For e.g., in a shopping cart application, let us consider a table which stores all the items (ITEM Table). The most common search pattern could be 'search by model name and price'. We can create an index based on the two columns, 'model name' and 'price'.

Full Table scan:

Full table scan is an operation on a table, which inspects every record of a table to find the record with the matching criteria. Full table scan is an expensive operation and should be avoided. Any search operation on non-indexed columns would result in a full table scan.

How does an index help in faster retrieval of data?

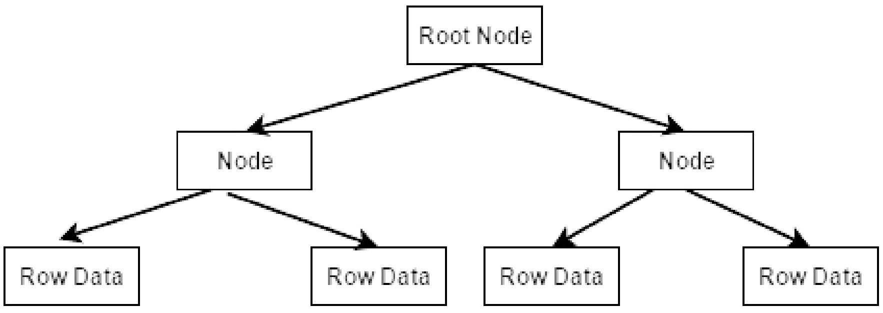
An index is stored as a data structure internally, typically as a sorted Binary Tree. The data structure itself contains index column details and pointers to actual data. So if we are searching for an item with name 'TV' and price '30k', the indexed search traverses the B-Tree and locates the node and then retrieves the address of the actual data. Thus it can get to the data node faster (B- Tree traversal is faster when compared to a full table scan)

Types of indexes:

Following are a few types of indexes which are commonly used:

1. Clustered Index
2. Non Clustered Index
3. Hash Index
4. Bit Map Index

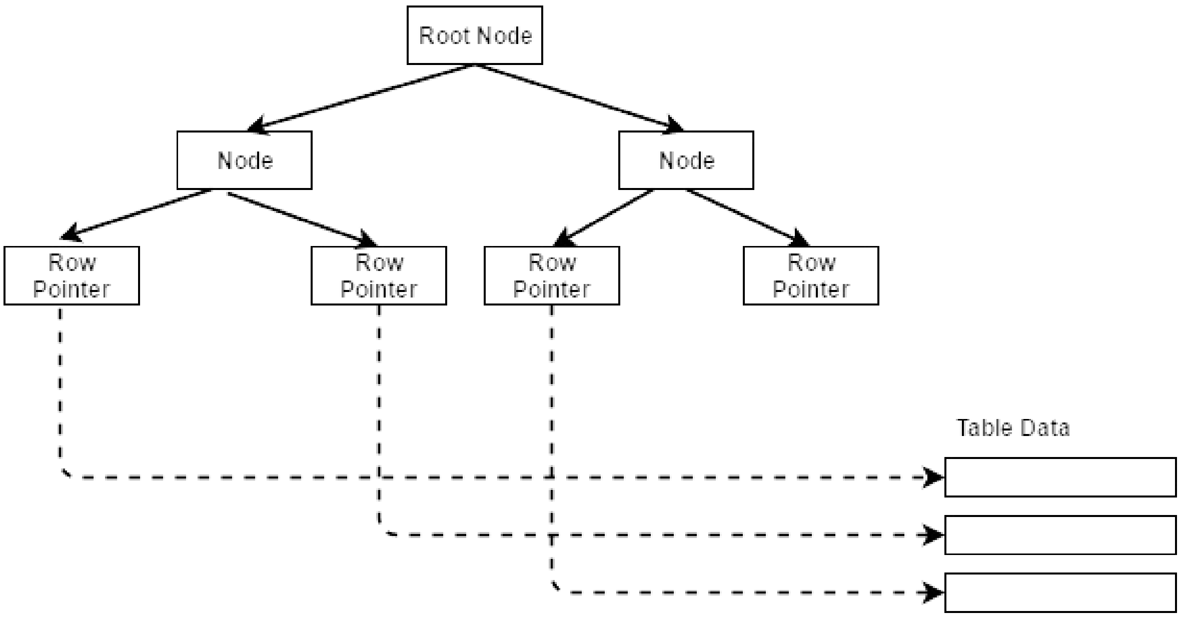
1. Clustered Index



Clustered Index

In this type of index, the data itself is stored in a sorted order as specified by the index. Since the data itself will be rearranged, there could be only one clustered index on a table. Clustered index helps in reducing I/O and faster access to data in a sequential order. (In most relational databases, the primary key will be a Clustered Index). As shown in above diagram, the table data itself is stored in a sorted order as a Balanced Binary tree.

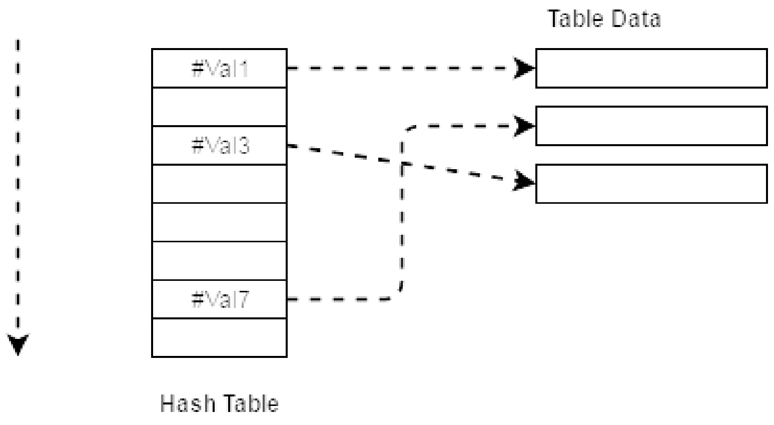
2. Non Clustered Index:



Non Clustered Index

In this type of index, the data itself is not stored in any particular order, but the index data is stored separately. The index data contains pointers to the actual data. As shown in the above diagram, the actual data is stored separately and the index data is a stored as a sorted Balanced Binary tree with the leaf nodes having a pointer to the actual data.

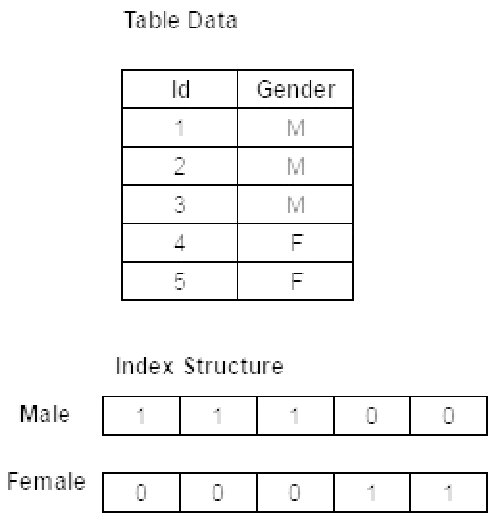
3. Hash Index:



Hash Index

In this type of index, the hash of the indexed columns is stored in a hash table. This type of data structure doesn't support inequality searches. For e.g., search like 'price >30000' cannot be performed using this index.

4. Bit map Index:



Bit Map Index

This type of index is generally used on columns which have low cardinality. i.e. the column should have few unique values. For instance, data like 'Gender', flag values like 'True/False' etc. could be considered for this index. A bitmap index stores a data structure for each unique value and sets the bits to 0/1 for every record based on column values.

Limitations of Index:

Indexes have their own limitation. Having too many indexes on a table would mean additional storage requirements to store the index data (in case of non-clustered index), and also might affect performance of insert/update statements as the index trees (or the data itself in case of clustered index) need to be reorganized in sorted order. So indexes have to be created judiciously, after thorough analysis.

Reference Implementations:

Most of the relational databases like Oracle, MySQL, and PostgreSQL implement one or the other above mentioned indexes and some of them also offer other variants of index implementations.

OLTP vs OLAP

OLTP:

In this section, we will take a look at two categories of database systems namely OLTP and OLAP.

OLTP:

OLTP (Online transaction processing) system, as the name suggests are transactional in nature. i.e. these systems involve read/write operations of live customer data. In such systems, generally, the storage (database) should be ACID in nature.

For instance, online banking system is an online transaction processing system, since it involves operations on live customer data.

OLAP:

OLAP (Online Analytical processing) systems generally operate on offline data. i.e. this data is generally a copy of the live data. Various statistical analyses are carried out on this data (also referred to as historical data) to extract value and meaning out of the data. These systems are also referred to as data warehouses.

The data is loaded onto these systems using batch processing or bulk operations like ETL (Extract, Transform and Load). Once loaded, these systems are read only in nature. The analytical operations derive meaning out of data, but do not modify the data. So these systems should be designed to be read efficient.

For instance, to analyze the banking customers spend patterns, the live customer data could be copied over to another system (OLAP system) to run the statistical analysis.

Transactions

In database systems, transactions denote a boundary within which a certain number of tasks are performed as a single logical unit. At the completion of a transaction, either all tasks are successful or all tasks are not successful.

Traditionally transactions are said to be ACID in nature. Below is the list of ACID properties which define a transaction.

1. Atomicity:

Atomicity ensures that each operation is carried out in its entirety or is not performed at all and the operation cannot be done partially. This, all or nothing approach is essential to ensure that a transaction either succeeds or fails. For example, transferring of money from one account to another consists of two steps namely, deducting amount from source and adding amount to target account. Both of these operations will be part of a transaction, and both operations should succeed for the transaction to complete.

2. Consistency:

Consistency ensures that at the end of each operation, the system is left in a valid, consistent state. i.e. the changes conform to all the rules, constraints and triggers etc. which are put in place in the system. For example, if creation of account is an operation, then there could be a database rule which specifies that the account number be unique. At the end of the transaction, this rule should be satisfied in totality.

3. Isolation:

Isolation ensures that each transaction is executed in its own execution context and is not impacted in any way by any other transactions running concurrently. This way, different transactions do not step on each other. Isolation levels provide control over transaction visibility and it affects the integrity of a transaction in various scenarios. There are four Isolation levels which could be chosen in a database system, depending on the use case:

- Read Uncommitted:
- Read Committed
- Repeatable Read
- Serializable

Before explaining these Isolation levels, let us take a look at few related concepts:

1. Dirty read:

Dirty read happens when one transaction is allowed to read uncommitted changes from another transaction. It is called as dirty read, since the changes are not permanent and if that transaction is rolled back, then those changes are of no value. For example, in the money transfer scenario, at the point of time, when the money is deducted from the source, but not yet added to the target account, the account balance in the source account should not reflect the deducted money. Because, there is a chance that the second operation might fail and transaction could be rolled back.

2. Non Repeatable read:

A non-repeatable read happens when multiple reads to a data or a record, results in differing data within the returned collection of results. This happens when another transaction is updating the data in between the reads. So we are actually looking at stale data in between updates. For example, while browsing through a list of articles for sale in an online application, it is possible that the article details are updated as we browse and the description which is seen a few moments back could have been updated when a moment later.

3. Phantom read:

A phantom read happens when multiple reads to a data or a record, results in different collection of results. Unlike dirty read, here the size of the collection returned itself might vary, since new records would have been inserted by other transactions. For example, in a shopping cart application, while browsing through the available products in stock, the products might disappear (or new products appear) as we navigate back and forth. This happens, since other users could be ordering those products at the same time (or new products could be added into stock). Phantom read could be an unavoidable scenario and might be a desirable behavior most of the times.

The below table shows how do the Isolation levels impact the read related issues.

	Dirty Read	**Phantom Read**	**Non Repeatable read**
Read Uncommitted	Yes	Yes	Yes
Read Committed	No	Yes	Yes
Repeatable Read	No	No	Yes
Serializable	No	No	No

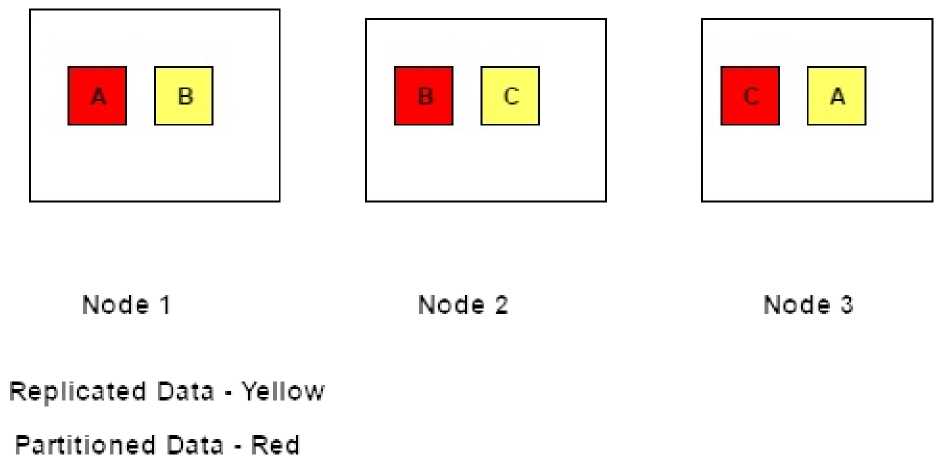
4. Durability:

Durability ensures that once a transaction is committed, the data is persisted permanently guarding against power loss, crashes, network failures etc.

Note:

Serializable is the highest level of Isolation level which provides very strict control, but it severely affects performance. Read Committed is generally considered a reasonable Isolation level.

Replication and Partitioning



Replication and Partitioning

In this section, we will take a look at two concepts which are generally used while scaling out data storage in a cloud environment. Consider a data storage system (any database), which needs to be scaled out for high availability in a cloud environment.

Replication:

In order to be highly available, the data needs to be stored in multiple nodes, so that if one node goes down, the other available nodes cater to the data requests. This process of having multiple copies of data is referred to as ‘Replication’.

Replication Factor:

A replication factor is the number of replicas (or copies) which would be maintained for each set of data. In the above diagram, the replication factor is two, since two copies are maintained apart from the primary data.

Challenges of replicating data:

Replication of data poses a very important challenge namely syncing of data. The changes made to any one copy of the data should be synced up in other nodes as well to guarantee consistency of data.

Partitioning:

Replication allows multiple copies of data to be available in multiple nodes. However, what if the data grows in size and one node is not capable of storing all the data. In such a scenario, we would need to split the data into multiple nodes, so that collectively those nodes account for the entire data. This process of splitting the data into multiple nodes is known as 'Partitioning'.

Challenges of partitioning data:

When we partition the data, the system needs to maintain book keeping of which part of data is stored in which node. Based on this information, the system needs to query the corresponding node when a request is made, in order to guarantee high performance.

Replication and Partitioning combined:

Replication and Partitioning could be combined for high availability and scalability from a data perspective. Thus we can have the data partitioned into multiple nodes and each partitioned node could be replicated across multiple nodes.

For instance, in the example above, the data (A, B, C) is partitioned into three nodes, with each node having one set of data. Also, the data is replicated across all the three nodes, in such a way that the partitioned data and replicated data are present in a mutually exclusive manner in any node. i.e., Node 1 maintains partitioned data A and replica of data B, Node 2 maintains partitioned data B and replica of data C, Node 3 maintains partitioned data C and replica of data A.

In such a setup, if node 1 goes down, the system would retrieve the data A from the replica in node 3.

Reference Implementations:

Cassandra database provides partitioning and replication of data. Mongo DB also provides partitioning through partitioned collections.

CAP Theorem

In this section, let us take a look CAP Theorem, which is a very important concept in distributed architectures.

From the WIKI, CAP Theorem (proposed by Eric Brewer), states, that, “It is impossible for a distributed computer system to simultaneously provide more than two out of three of the following guarantees: Consistency, Availability, Partition tolerance”.

Following definitions are from WIKI:

1. Consistency:

Every read receives the most recent write or an error. The data is always consistent, regardless of situation.

2. Availability:

Every request receives a (non-error) response – without guarantee that it contains the most recent write.

3. Partition tolerance:

The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes. A system is said to be Partition Tolerant, if the system behaves normally, when there is a network failure between nodes so that they don’t communicate with each and there is a partition between the nodes.

Revised CAP Theorem:

Nearly a decade after proposing the CAP Theorem, Eric Brewer proposed a revised CAP Theorem, according to which, network partitions cannot be avoided, and the distributed system should tolerate the partition. In the absence of partition, the system could support Strict Consistency and Availability. In the case of a network partition, the system should handle the partition and should recover from that. During a network partition, we only have two choices, either be Available or be Consistent.

What is the relevance of CAP Theorem?

Before we start looking at CAP Theorem in more detail, let us take a look at why did CAP theorem arise in first place.

Traditionally, databases were ACID in nature. They were quite happy being strictly consistent and serving loads which were moderate (relative to today's context). However, the past few years saw the emergence of social media, increased internet usage and cloud based applications. Thus, the systems started experiencing huge workloads and systems are expected to be available 365X24X7. This requirement of being 'Highly Available', gave rise to new architectures. So, being consistent alone, ceased to be a criteria for modern databases. If we need systems to be highly available, we cannot achieve this with one machine (or even a couple of machines). We will need a distributed architecture, with data being replicated as well as distributed (Partitioned), across multiple machines.

Use of multiple machines (or nodes) to store data, and use of commodity hardware, means increased chances of failures. It also gave rise to increased complexity of failover mechanisms. Questions like 'What if one node goes down?, Will the system function normally? How are the data operations like read and write affected' began surfacing quite often.

Naturally, in such scenarios, we need to trade over one factor for the other. CAP Theorem proposes one such tradeoff for distributed data architecture.

CAP Theorem Scenarios

Since CAP Theorem states that any two of the three (Consistency, Availability, Partition tolerance, referred to as CAP) factors could be guaranteed, we can have three possible scenarios.

1. AP (Availability, Partition tolerance)
2. CA (Consistency, Availability)
3. CP (Consistency, Partition tolerance)

Let us consider a distributed architecture, which has multiple nodes which has data replicated and distributed across multiple nodes. Let us consider two nodes which have data distributed across them. Consider a network failure between these two, which prevents communication between these two nodes.

1. AP (Availability, Partition tolerance)

In this combination, the system is not strictly consistent, when there is a network partition. Now, if we allow the system to update the data on either of the node, the other node will not get updated. Thus, the system is still allowing data operations, thus being Available.

However, the data is not consistent. i.e., one node has latest data, whereas the other node has stale data, leading to consistency issues.

2. CP (Consistency, Partition tolerance)

In this combination, the system is not available, when there is a network partition. In the above scenario, if the system doesn’t allow data operations and the nodes return an error, when there is a network partition, the system will be still be Consistent. But this comes at the cost of Availability.

Also, when there is a network partitioning, the ACID properties could be guaranteed within each partition. For instance, the operations could be Atomic, Consistent within a partition, but not necessarily across the entire system which is distributed.

NoSQL Databases

NoSQL is a new class of databases, which are very different from the traditional Relational databases (also known as RDBMS).

Before diving into the details of a NoSQL database, let us take a look at the shortcomings of a RDBMS and the benefits of using a NoSQL database.

Why do we need a NoSQL database?

1. Flexible Schema:

In the RDBMS world, the schema needs to be defined or designed upfront. Schema definition includes tables, columns, column data types, column data length etc. This could be a hindrance, if the format of the data which is stored changes frequently (due to varying reasons like, frequently changing requirements to align with rapid changes in business, or because the data is not in our control and the source of data is elsewhere). For instance, let's say we are designing a system which captures work experience of people. There could be a vast variety of data that could be potentially captured and it may not be possible to define the schema upfront.

Another issue with rigid schema is that, we end up paying for storage which we don't use. Let's say there is an attribute which is used only 10% of times. In a RDBMS, whether the column has data or not, it still consumes storage. So we are wasting storage space and even paying for it!

NoSQL databases on the other hand support flexible schema. (To draw parallels with RDBMS, imagine having to define only a table and then, columns could be defined on the fly as the data comes in). If an entity has 20 different attributes and each of these attributes have data for only 10% of the entire data, then it consumes space for only those 10% of the data. Thus we get flexibility as well as efficient use of storage.

Name (John)	Age (30)	
Name (Kelly)	Age (23)	Salary (60k)
Name (Howard)	Age (28)	

In the table shown above, only one record (having name Kelly), has a salary attribute, whereas the other records do not. The salary attribute occupies space only for that record.

2. Availability:

Traditional RDBMS generally focus on data integrity and reliability and thus may not be highly available. Also, they are not distributed by design. Though it is possible to have distributed RDBMS databases, they are generally expensive and involve complicated setup and configurations. Most of the NoSQL databases are generally distributed by design and are comparatively inexpensive. Since they are distributed, they are highly available.

With the advent of cloud, high availability is an essential characteristic of most applications. So it is no surprise that NoSQL databases are gaining popularity.

NoSQL database types:

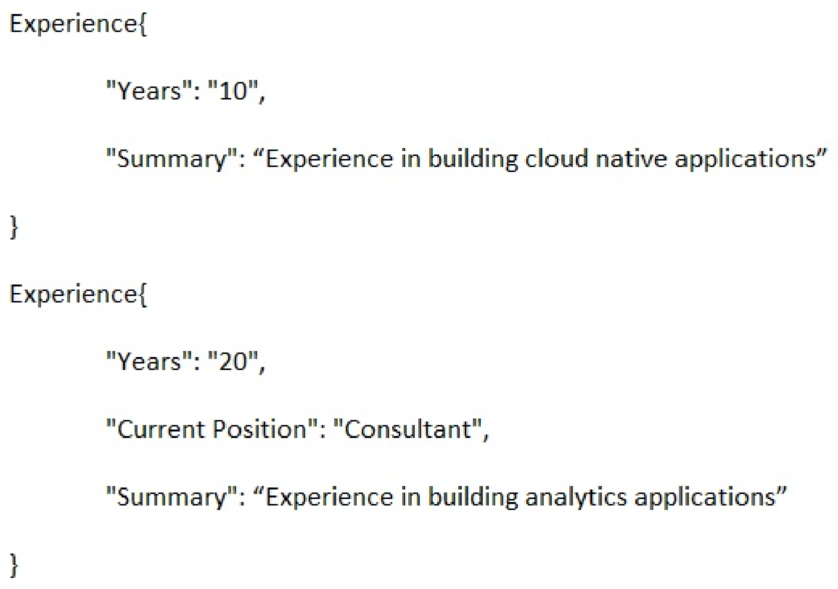
It is important to have a good understanding of the types of NoSQL databases and the pros and cons of each, so that we are in a position to choose the right database for the right job.

Types of NoSQL databases:

- Document Based databases
- Key Value databases
- Columnar Based databases
- Graph Based databases

1. Document Based Databases:

These types of NoSQL databases store and retrieve document format of data. A document could be a XML document, a JSON document, a YAML or a BSON document. Each document will have field name and values encoded in specific format. For e.g., XML document will have element name and value, JSON document will have attribute name and value etc. The document will be stored and retrieved as is. Also, each document data layout could be different from the other, providing flexibility.



```
Experience{

        "Years": "10",

        "Summary": “Experience in building cloud native applications”

}

Experience{

        "Years": "20",

        "Current Position": "Consultant",

        "Summary": “Experience in building analytics applications”

}
```

In the above sample document, the second ‘experience’ document has an attribute ‘Current Position’ which is not present in the first one and this provides flexibility. Some of these databases also support search by individual fields and support indexing by fields.

Reference Implementations:

Mongo DB, Couch DB, Elastic Search

2. Key-Value Databases:

These types of NoSQL Databases store data using a data structure like ‘Dictionary’ or a ‘Hash’. A Hash stores data in terms of keys and values. A value could be either a single value or a tuple (list of values) or even another hash. The key should be unique in a given Dictionary and the hash uses the 'Hashing Algorithm' to retrieve the values.

Key	Value
Car	Honda, Toyota, Mercedes
Bike	Harley Davidson, Kawasaki

In the above example, the key ‘Car’ has values like ‘Honda, Toyota, Mercedes’.

Reference Implementations:

Redis, Riak DB

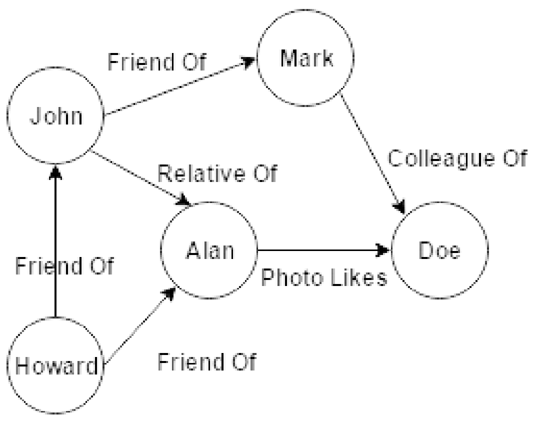
3. Columnar Database:

In this type of NoSQL database, data is grouped into columns rather than into rows. Columns are grouped logically into column families. The data is partitioned and stored based on columns. Column based partitioning supports aggregation functions quite easily. For instance, let us say we want to compute the average age of a customer from a customer table having a column called ‘age’. In a RDBMS, the average function would have to scan every row and perform aggregation. This would include a full table scan and is a very costly operation. On the other hand, in a columnar database, all the values of a given column are stored close to each other. So it is faster to perform column specific aggregation function.

Reference Implementations:

HBase, Cassandra

4. Graph database:



Graph Database

These types of databases store data as nodes and edges of a connected graph. Nodes represent data and edge represents relationship between nodes. Graph databases are useful in representing data where relationships between data points also need to be represented. A Graph database provides queries which helps in traversing the graph in a simple and efficient manner.

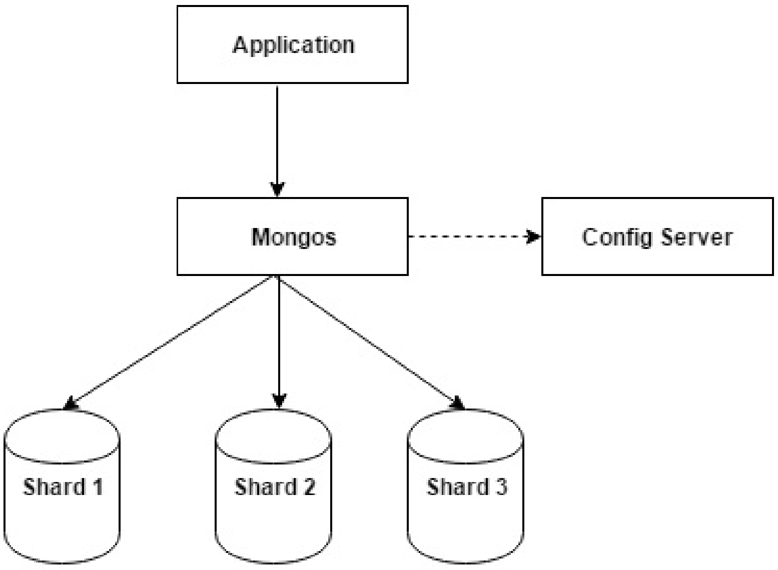
For e.g., in a social media application, a Graph database could be used to connect people in a network.

Reference Implementations:

Neo4J

NoSQL Reference Architectures

Mongo Database Architecture:



Sharding in MongoDB

Sharding

This is a method of horizontally partitioning data in a database. This concept is similar to Partitioning explained earlier. Each individual partition is referred to as a 'Shard'. MongoDB distributes the read and write workload across the shards and provides horizontal scalability.

Mongo DB will distribute data into different shards based on a shard key. A shard key is a field which should exist in every document of a collection.

For instance, in a system which stores student information across US, State could be a Shard Key. MongoDB provides two components to allow data distribution:

1. Router (known as Mongos).
2. Config Server

The config Server would track and store the metadata. It also stores information on which shard key is stored in which shard.

The Clients should never connect to individual shards, but always should go through Mongos. The Mongos router would cache the metadata information and would route the query request to the appropriate shard. If a shard key is not included in the query request, Mongos would broadcast the query to all the shards.

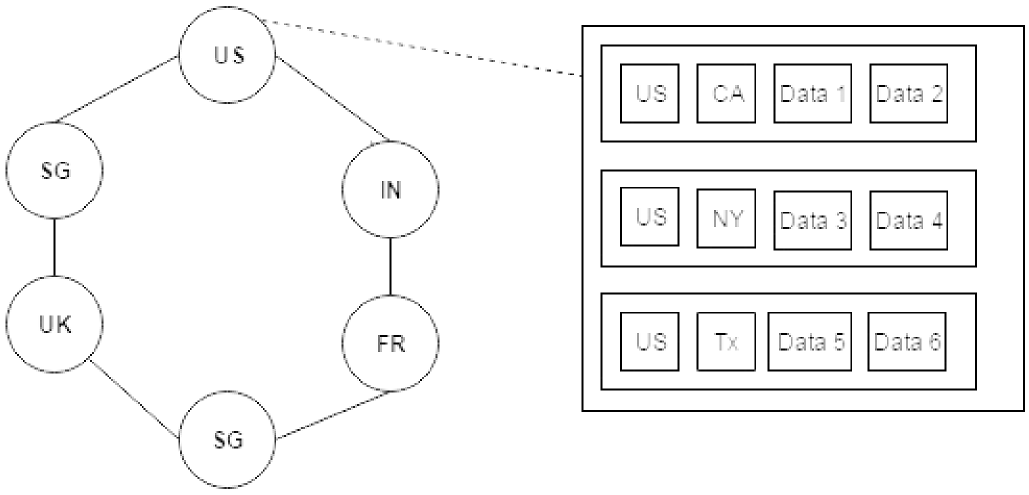
Redis Architecture:

Redis is an In-memory, Key value based database. Redis stores all the data in memory (RAM), so it provides faster access to data. But we need to keep the memory limitation in mind. One of the use cases where Redis could be used is in 'session persistence'. Another use case is to use Redis as a distributed cache (to cache frequently used data).

Redis supports persistence of data to the disk. Redis could be configured to store data to the disk at regular intervals.

Redis also supports Replication (Master/Slave) and Clustering (Sharding) modes of configuration.

Cassandra Architecture:



Cassandra Architecture

Cassandra has a distributed architecture, where data is distributed among multiple nodes and is also replicated to handle fail overs. Cassandra has a peer to peer distributed system, where in, the clients can fire a query by connecting to any node in the cluster. That node would be responsible to coordinate with other nodes and return the results. (Note that there is no Master node in this setup). So every node would be aware of the metadata and configuration (nodes communicate with each other using a protocol known as 'Gossip Protocol'). A Cassandra cluster could be generally visualized as a ring consisting of multiple nodes where data is distributed consistently across all the nodes.

Cassandra is a row oriented database. Every table has a Partition key (or a combination of keys) and a Clustering key (or a combination of keys).

Partition Key decides which partition the data would go to. Clustering key decides which cluster in a given partition, the data resides and helps in sorting the data in a partition.

For instance, in the above diagram, let us say the country code is the partition key and state code is the clustering key. Each partition would get data depending on the hash of the country code. So one partition would get US, another partition would get IN etc. Within each partition, the data would be sorted using clustering key which is the state code.

So when a row is inserted into a table, a hash function of the partition key is calculated and this hash function would decide which node would get that partition. Similarly, while performing a read operation, based on the partition key provided in the read operation, the specific node which contains that partition would be queried.

For this reason, it is very important to have a data model built around querying patterns in Cassandra. If the partition key changes often, then using Cassandra is not a good fit. Also, the queries must have a partition key to avoid querying all nodes and avoid scanning all partitions.

Cassandra is generally a good fit for heavy writes since it performs very well on writes.

Big Data

In this section, we will take a look at Big data and the concepts related to Big Data. Big data has become very relevant and gathered all the attention, mainly due to the rise of internet usage and corresponding explosion of data being collected by various organizations. Data is generated by various devices, by human or by machines, generated at real time. Now that organizations have data from various sources, they would naturally want get insights from data which would help them take business decisions.

What is Big Data?

Big Data is a terminology which is associated with computation and analytical processing of three forms of data.

1. Volume
2. Variety
3. Velocity

Before, we go deeper into these three forms of data; let us understand what does big data actually deal with. Big data involves performing analysis of data to derive value or meaning out of data. This is partly similar to what traditional analytical systems (also known as data warehouses) used to do. They analyze historical data and answer certain questions on the past behavior (of users, systems or whatever). However, Big data systems go one step further. They also predict the future behavior by building models from the historical data. For instance, if traditional systems were capable of answering questions like “In which quarter of the last year, did the maximum sales of a car happen?” the modern Big data systems would also be capable of answering questions like “In which quarter of the next year, can the maximum sales of a car happen?”

Big data systems also derive patterns from data by running various algorithms on the data. Though, many of these tasks were also performed earlier, what changed from the traditional systems is the above 3 Vs.

Let us take a look at each of these V factors.

1. Volume

Big data deals with huge volumes of data in the range of excess of peta bytes of data. These volumes are not something which could be handled easily by traditional data storage or data processing systems.

2. Variety

Big data deals with different types of data and not necessarily the data which would fit into SQL driven relational data. Traditional data warehouses, generally loaded data into Relational databases and ran SQL queries or procedures. However, the data could have originated from a variety of sources and the data format could be in unstructured text format, or media content like images, videos etc. Following are a few examples of data from different sources.

- Data from social networking platforms like Twitter, Facebook etc.
- Data from Internet connected devices like sensors, devices in cars etc.
- Data collected from users surfing the internet, like websites visited, IP address of the users, time spent on each site etc. (which is generally referred to as Click Stream data).
- Data collected from surveys like health survey, employee survey etc.

3. Velocity

Big data deals with data arriving at very high velocity or speed. Traditional analytical systems dealt with data which was at rest (copied versions of real time data). However, Big data also deals with data which changes in real time. For instance, let us consider a real time fraud checking system in a banking application. The system has to recognize that a transaction is fraudulent in real time (the response time should be a few seconds) and prevent the transaction.

Tools and techniques for Big Data

The 3 Vs of Big data simply means that the traditional databases and the data processing algorithms would not be able to support it one way or the other. Obviously, we need modern data storage systems, algorithms and tools to work with Big data. Also, Big data storage and processing systems need to be distributed in nature in order to scale.

Modern Data Algorithms:

To process and analyze data in Big data systems, a lot of modern algorithms are available, which may not be supported by traditional database systems. Some of the algorithms include Map Reduce, various Machine learning algorithms etc.

Data storage systems:

Data storage systems like Hadoop, No Sql databases like Cassandra, In-Memory data processing systems like Apache Spark support various features, algorithms necessary to operate on Big data.

Data processing types

While discussing Big data, we will take a look at two different categories of processing data.

1. Stream Processing
2. Batch processing

Let us take a look at each of these.

1. Stream Processing

Stream processing is a programming paradigm where data arrives to a system continuously as a stream and the data needs to be processed in real time. Stream processing requires high performance in-memory processing systems. Libraries like Apache Spark provide Stream Processing abilities.

2. Batch Processing

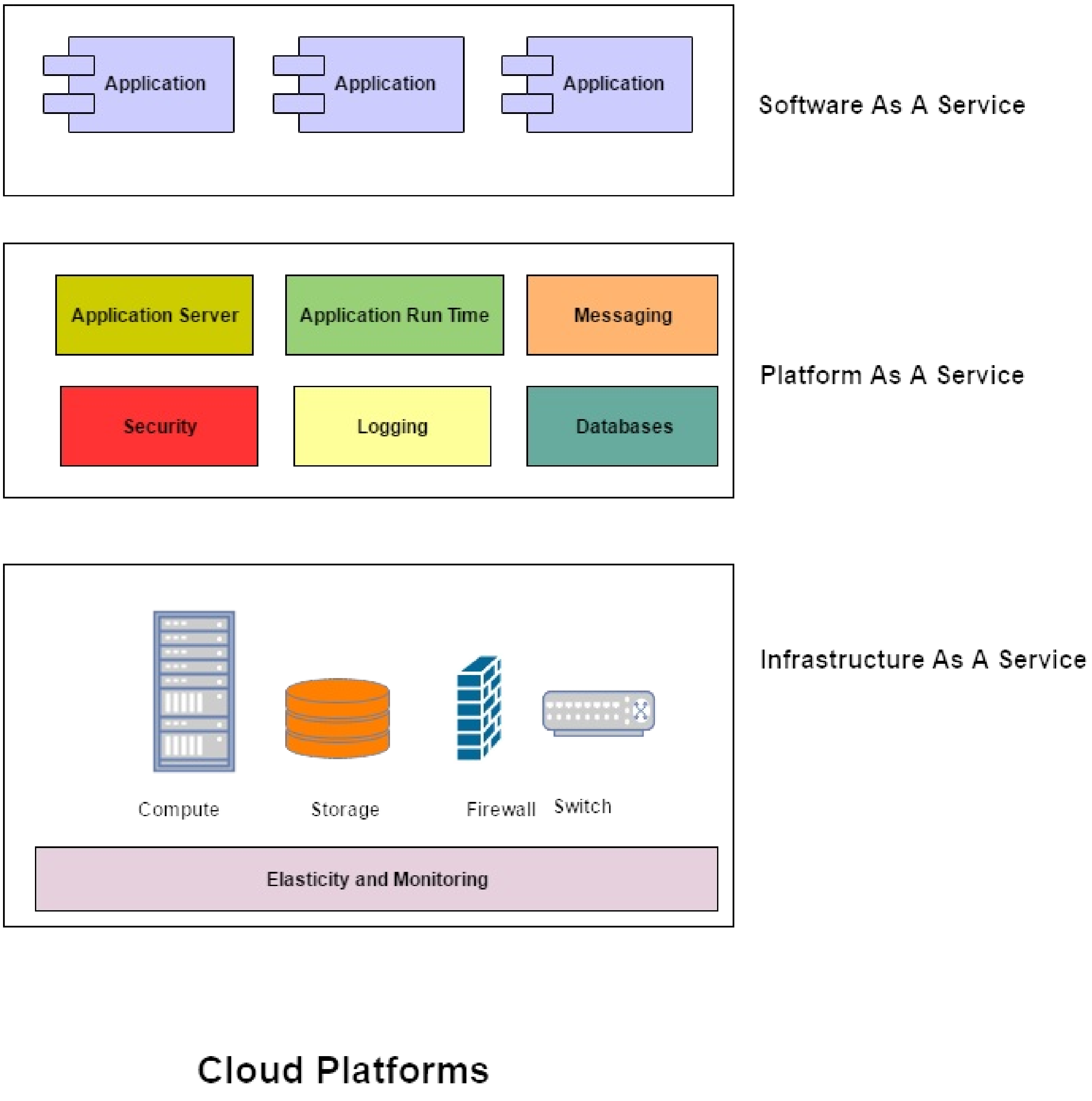
Batch Processing is offline data processing of huge data for carrying out analytics or import/export functionalities. In contrast to Stream processing, Batch processing operates on data at rest. For analysis of big data, map reduce techniques are suitable to provide desirable results.

Cloud Computing

This section explains the various Cloud Computing concepts in detail.

Cloud Platforms

In this section, we will take a look at the different types of Cloud Platforms and their characteristics.



There are three types of cloud platforms:

1. Infrastructure as a Service (IAAS)
2. Platform as a Service (PAAS)
3. Software as a Service (SAAS)

1. Infrastructure as a Service (IAAS):

This type of cloud platform provisions and configures infrastructure and provides infrastructure on demand. For e.g., it might provide you a pre provisioned virtual machine or a firewall with preconfigured rules. If we are going to scale our application, we are going to need more machines with some specific configurations. The platform will manage the infrastructure for us.

Traditional datacenter vs IAAS:

Traditional datacenters used virtualized servers or physical servers to host applications. These datacenters had fixed hardware capacity and it was not easy to add more machines. Also, addition of resources was a manual process. Setting up a new datacenter generally is a herculean task which could take months altogether.

An IAAS, on the other hand, would make the task of provisioning new machines, creating new setup extremely easy, since the IAAS provides well defined APIs to configure and setup resources. An IAAS could either be a private cloud setup or a public cloud setup. In either scenario, the cloud environment provides easy scalability and quick setup. The cloud environment provides the typical cloud characteristics as described later in this chapter.

Reference Implementations:

AWS, OpenStack, VMWare VSphere

2. Platform as a Service (PAAS):

This type of cloud platform provides and manages the runtime environment required to run applications and also manages the application itself. It also provides an environment for application to run along with all the dependencies like database, queues etc.

A PAAS layer sits on top of the IAAS layer and abstracts away the infrastructure management.

Traditional non PAAS environments vs PAAS environments:

Let us take a look at traditional non PAAS environment. Let us consider a Java based web application which would need a PostGres SQL database and a message queue system. To deploy this entire application, we would typically need to install and configure the following systems:

- Java Virtual Machine
- Web application server (like Tomcat)

- Load balancer
- Database (like PostGres)
- Messaging system (like Rabbit MQ)

If these systems are located in different nodes, then we need to configure connectivity between these systems and also configure the load balancer to route the request to different nodes. If we have to scale the application by increasing the number of instances, then we would need to perform the following steps:

1. Provision a new virtual machine (if we are using a IAAS, then provision using that)
2. Install web application server on the new node (or machine)
3. Deploy the application to the new machine
4. Reconfigure the load balancer.

Clearly, it is a tedious process. A PAAS environment takes care of most of the above tasks and allows the developer to focus on building application logic. For the above example, let us see how a PAAS helps.

When an application is pushed to a PAAS:

1. PAAS would identify the application (whether it is Ruby, Python or Java) and provide a corresponding runtime environment.
2. PAAS would provide the corresponding application server required to run the application.
3. PAAS would configure the load balancer with the end point provided while pushing the application. We can easily scale the application to have more (or less) instances, with single command or button push in the UI.
4. PAAS provides backing services like Database, Message queues, which we can associate with the application, with simple commands or UI. These database, queues are managed by the PAAS itself.

Extensibility of PAAS platforms:

PAAS platforms are generally extensible to add new run times, new applications servers (for example, if we are using a new run time, which is not supported by the PAAS out of the box). PAAS platforms also support integrations with proprietary databases or soft wares which are not hosted and managed by the PAAS.

Reference Implementations:

Cloud Foundry, OpenShift

3. Software as a Service (SAAS):

This type of cloud platform provides specific software products/features as a service and allows for configuring/customizing the service based on customer requirement. Software As a Service platform is generally used by product based companies who deploy their products on the Cloud platform and customers can sign up and use those hosted products. Instead of deploying these products within their premises, customers can use the hosted solutions. Software As a Service would generally have a PAAS, or an IAAS underneath it, in order to achieve scalability and availability.

Traditional software vs SAAS:

Before the concept of SAAS came into picture, organizations which were selling software had two options in which they could sell software:

- Deploy on Premise: The customers need to deploy the software behind their firewall. This option would mean, the installation, configuration and maintenance would take place in the customer's premises and the underlying infrastructure has to be maintained by the customer. For example, the CRM, ERP products sold by major companies followed this model. The disadvantage with this model is that, the customer has to ensure high availability of the infrastructure.
- Hosted solution: The software is hosted by the organization which built and sold the software, and this software is managed, configured, maintained by the seller organization on virtualized infrastructure (there is no cloud platform here). However, this solution is not scalable, and if the number of customers increase, then the host organization has to expand its infrastructure and it could take months to add new infrastructure. Also, it is not easy to guarantee high availability of the software and also it is not easy to measure and add a pricing per module, as in Pay per Use models.

A SAAS solution, on the other hand, is similar to the 'hosted solution' explained above. The significant difference is that, the software is deployed on a cloud infrastructure which is scalable, resilient and exhibits all the characteristics that a cloud platform should have. Also, since it uses a cloud platform, it is easier to measure the number of API invocations or service calls and hence it is easier to bill the customer per service. This model known as 'Pay per Use' model is commonly used by SAAS providers. For instance, a customer might have access to various modules of a Customer Relationship Management solution, but the customer would be charged only for those modules which are used in a given month. This option works out to be financially conducive to customers and also relinquishes the pain of managing infrastructure if they had deployed within their premise.

Reference Implementations:

Salesforce applications like CRM, Atlassian Jira

Characteristics of a Cloud Platform

There are three categories of cloud namely:

1. Private Cloud
2. Public Cloud
3. Hybrid Cloud

Let us take a look at each of these in detail.

1. Private Cloud: (On Premise Cloud)

This type of cloud platform will be hosted behind the corporate firewall and will be managed by the organization.

Reference Implementations:

Cloud platforms like Cloud Foundry, Open Stack offer On Premise installations of Cloud. If the customers chose to use these kinds of cloud solutions, then they will have to install and maintain these environments on their own.

2. Public Cloud:

This type of cloud platform will be hosted on the internet and will be managed by one company. Each customer will be provisioned on the same infrastructure, providing multi tenancy.

Reference Implementations:

Cloud platforms like Amazon Web Services (AWS), Google Cloud Engine, and Digital Ocean offer public cloud services. These environments are maintained by the respective companies and the customers only pay for the cloud services/resources which they consume.

Public Cloud Vs Private Cloud:

With public cloud solutions, customers need not worry about the headache of maintaining cloud environments and they can simply use the services offered on a pay per use model. Also, in a private cloud, the scalability depends on the initial sizing of the cloud environment. For instance, while setting up a cloud environment on premise, let us say the organization estimated that it would cater to 5000 CPU workloads each with 1GB RAM. It is possible that within one year, the company expanded and grew to an extent that, these workloads are no longer sufficient. In such a case, the company has to invest in resizing and expanding its cloud environment. Depending on the private cloud solution which they have chosen, this may or may not be an easy task. So, the initial capacity sizing of the cloud infrastructure is very important and has to factor in the vision and growth predictions of the organization.

3. Hybrid Cloud:

Hybrid cloud is a mixture of Public and Private Clouds, with some resources residing behind company firewall (private) and some on the internet.

Reference Use cases:

Some companies would want to use private cloud for storing data on premise (to ensure data security and probably for legal compliance). However, they would also want to use public cloud resources to host applications, which operate on this data. They would want public cloud resources as they often might want high scalability for their operations, which might be difficult to achieve with private cloud. In such a scenario, they can go for a mixture of both private and public cloud environments.

Characteristics of a Cloud Platform:

Following are some of the essential characteristics of a cloud platform.

1. Self-Healing
2. Multi-Tenant
3. Elasticity
4. Ability to Meter services

1. Self-Healing:

A cloud platform should have the ability to recover from outages, servers going down etc. without manual intervention. A cloud platform should ensure that the virtual machines which are spawned or provisioned are up and running all the times. It is the cloud platform’s responsibility to maintain those SLAs.

2. Multi-Tenant:

A cloud Platform should support multi tenancy. Multi tenancy is a feature, which allows multiple organizations (tenants) to share the same infrastructure in a cloud platform. Generally, there is a misconception that only public clouds are multi-tenant. That is a misconception, since, even in a private cloud maintained by an organization, it is possible that multiple divisions of that organization share the same cloud platform. In such a scenario, the multi-tenancy applies to different divisions.

3. Elasticity:

A cloud platform has to be elastic. i.e., the cloud platform should have the ability to scale up or down on demand and provision resources on demand.

4. Ability to Meter services:

The cloud platform should have the ability to measure and monetize each and every service it provides to its consumers.

Factors to be considered while choosing a Cloud Platform:

Following are some of the salient factors which needs to considered while selecting a Cloud Platform:

1. Legal compliance:

Some countries have laws which mandate the data to reside in the same country, where its users reside. In such circumstances, if we chose public cloud, we have to ensure the data center of the public cloud lies in that country.

2. Security:

Public clouds are more prone to attacks and also risks issues with multi tenancy. i.e an inadvertent bug in the cloud platform or a rogue application hosted on that, might expose data of one tenant to another.

3. Cost:

There are two parts to the cost factor:

- Maintenance cost in private Cloud
- Metered cost of services in public cloud

4. Datacenter Maintenance

If the Cloud platform is maintained in-house, then the following factors need to be considered from maintenance and sustainability perspective:

- High availability
- Real estate issues -
- Uninterrupted power supply
- Disaster recovery

Infrastructure As a Service

In this Section, we will take a look at Infrastructure As a Service (abbreviated as IAAS).

A traditional datacenter generally uses virtualization to create virtual machines and then deploy or install software in those virtual machines. In this kind of an environment, adding new virtual machine on demand is not a trivial task. Also, if a virtual machine goes down, there are no mechanisms available to bring them up automatically.

The issues in a traditional datacenter could be summarized as below:

1. Not Resilient to failures
2. Manual creation and configuration
3. Cannot remove or add infrastructure on demand
4. Configuration of Firewall, load balancers etc., are done manually

How does IAAS help?

To put in a succinct manner, all of the above issues are taken care of by the IAAS platform.

The platform takes care of the following:

1. Creating and provisioning of virtual machines on demand
2. Resilient – If one virtual machine goes down, the platform takes responsibility to spawn another machine with same configuration and state.
3. Provides additional infrastructure, network, storage configurations like firewall, load balancers, security policies. All these configurations could be stored in a template and the template could be used repeatedly.
4. Provides multi tenancy. i.e. Provides isolated infrastructure for each account.
5. Code based Infrastructure creation (Infrastructure as Code) – The platform exposes APIs which would create and configure infrastructure. So we can create a template of the configurations (like CPU, Memory etc.) and invoke the APIs on our own to provision infrastructure.
6. Provides Elasticity – The platform allows us to scale the infrastructure components based on predefined rules. For example, if CPU utilization is more than 80%, we can have a rule to scale horizontally.

Reference Implementations:

Some of the popular IAAS platforms are Amazon Web Services (AWS), OpenStack, VMWare vCloud Air.

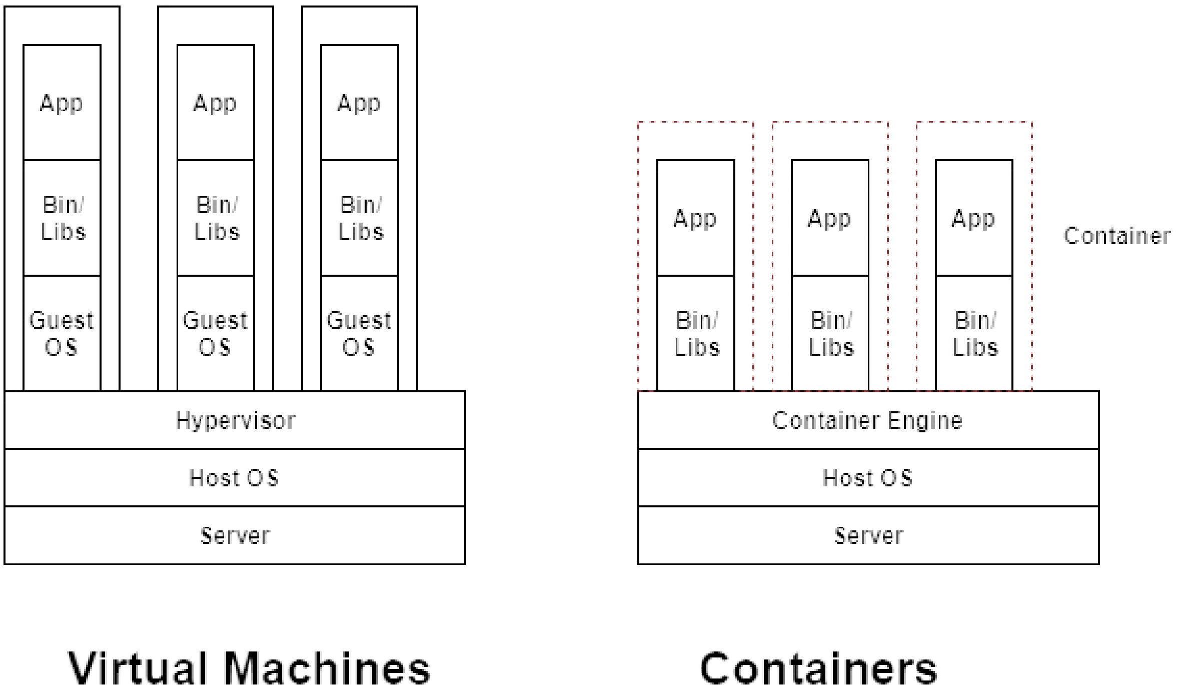
Containers

In this section we will take a look at Container technology, which has emerged as a key technology in shipping code and has had a huge impact on deploying applications on the cloud.

Containerization:

Containerization, also called container-based virtualization or application containerization, is an operating system level virtualization method for deploying and running distributed applications without launching an entire virtual machine for each application. Instead, multiple isolated systems, called containers, are run on a single control host and all the containers access a single kernel.

Containers Vs Virtual Machines:



The diagram above illustrates the difference between virtual machines and containers.

Virtual Machines:

Virtual machines use hypervisor software which is installed on a physical server having an operating system. Each virtual machine created, will have its own operating system (‘Guest OS’ in the above diagram). Thus the additional layer of OS on each virtual machine consumes significant disk space and adds overhead.

Containers:

A container on the other hand, does not require an OS of its own, but rather packages only those libraries and binaries which are required for the application to run. Also, the libraries which are common to many containers could be shared (it uses an immutable file system technology), thus making containers light weight. Containers provide an isolated execution environment for the application to run.

Why do we need Containers?

1. Avoids library conflicts

Using containers help us to eliminate library conflicts. Let us consider a scenario, in which we have to deploy multiple applications on a virtual machine and each of them is a Python application. If each of these Python application needs a different version of Python, then there is a high chance of facing conflicts in the library files, leading to nasty bugs or installation issues. Containers help overcome this issue, since the container would package each of the applications, its run time dependencies and the corresponding libraries independently and hence will be isolated with no interference from other containers.

2. Memory Isolation

If multiple applications are running on a virtual machine, it is possible that one application can advertently/inadvertently write into the memory area (RAM) used by the other application, leading to data loss, data theft, or corruption. Containers provide a separate run time environment for the contained application and it would not allow any other application outside the container to access it.

3. Ease of deployment across various environments

While deploying an application, there could be a series of complex steps to be followed to install the libraries including, dependencies, use of right version of libraries, managing the configurations etc. Containers help make these laborious tasks simpler, by creating an image through scripts for the above steps and the image could be installed in any environment. This removes installation or deployment errors and also eliminates manual interventions, thereby improving quality of overall deliverables.

Container Lifecycle:

Let us take a look at a typical container lifecycle. A container undergoes through a state transition as below:

1. Created – When a container is first created from an image, it will be in a state ‘Created’.
2. Running – When a container is started, it goes to the state ‘Running’. In this state, the applications in the container also start running (depending on the startup command provided which would be provided while starting the container).
3. Stopped – When a container is stopped, it goes to the state ‘Stopped’. The container could be started again, from the same container image.

Containers provide network interfaces to access the applications hosted within the container through an IP address and a port. Containers images could be uploaded to a registry (there are public registries or corporates could setup their own registry).

For instance, Docker provides a public registry, known as DockerHub, where we can find images of most of the commonly used soft wares.

Volumes:

Volumes in a container terminology are storage drives which could be attached to a container instance. Containers are generally stateless and only maintain the state in the RAM, while they are running. Once they are stopped, the state is no longer available. This works fine for applications (for example web application servers, which do not require state to be persisted). However, let us say, we want to run a database like Mongo DB in a container. In such a scenario, we will definitely need a way to persist the container state. Volumes come in handy, in such a scenario. Volumes allow us to attach a storage drive to a container and the application running in the container can write to the volume similar to they write to a local file system.

Why are Containers so important in Cloud?

With the advent of cloud, where, we will have several instances of an application in a cloud environment, it would be very tedious to setup the application runtime environments in all the machines. Also, in certain, cases, where we would need to switch deployment platforms, or switch cloud platforms, it could get extremely painful if we need to perform the environment setup all over again.

Containers come to our rescue here, since; we can containerize all our applications and deploy it in whichever environment we would like to. Since the application dependencies are packaged within the container, we would not have to bother about setting up environments again and again. Also, if there is some change to setup, we can just update the base image of the container and deploy the updated container image to all machines.

Reference Implementations of Container Technologies:

Some of the popular container implementations are Docker, Core OS.

Container Orchestration

Container Orchestration is a process of deploying and managing the life cycle of a number of containers in a given environment.

Why do we need Container Orchestration?

When we have a large number of containers, they pose a few challenges with operating and maintaining the individual containers, once the number of containers increases.

Following are the some of the common features provided by a Container Orchestration Engine:

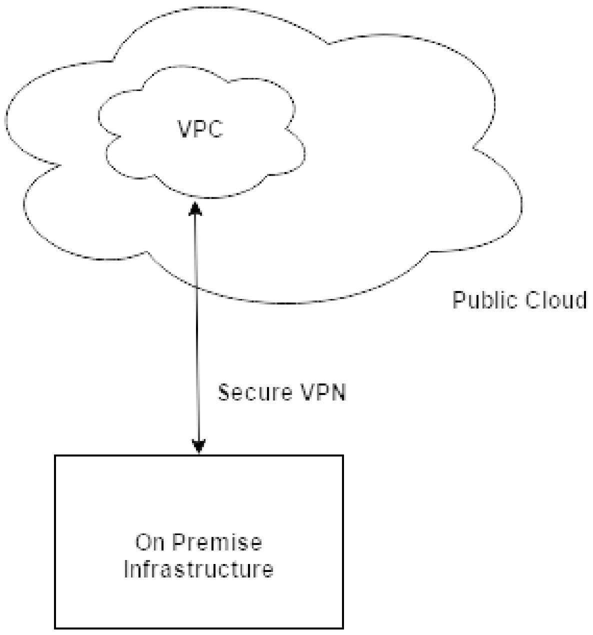
1. Managing lifecycle/deployment of Containers
2. Replication of application instances providing horizontal scalability: Each application instance could be scaled horizontally on demand with minimal effort.
3. Allow communication between containers (Security/Networking aspects): Each container has to be identified through an IP and the internal network communication across containers and appropriate network protocols (TCP or UDP) have to be supported.
4. Monitor the state of the containers and act on the containers (bring them up or down): Monitor container health and ensure that configured number of container instances is always running and any given time. For instance, if we configure 3 instances for an application instance, if any one instance goes down, the orchestration engine has to identify that the instance is down, and has to bring it back up, thus providing a self-healing feature.
5. Provide service discovery: Each container (or the application instance) has to be discovered uniquely through a name (internally through an IP), so as to enable communication with that container.
6. Load balancing across replicated container instances: When an application instance is replicated or scaled horizontally, we need to load balance the requests to that application across instances. The orchestration engine has to provide components which will perform this activity.
7. Manage configuration details of the applications

A container orchestration tool provides the above capabilities and abstract away the complex tasks of managing containers.

Reference Implementation of Container Orchestration tools:

1. Kubernetes
2. Mesos and Marathon
3. Diego for Cloud Foundry
4. Amazon EC2 Container Service
5. Docker Swarm

Virtual Private Cloud



Virtual Private Cloud

Public clouds generally offer a feature known as ‘Virtual Private Network’ or ‘Virtual Private Cloud’. Though the actual name could vary by the cloud provider, the concept remains the same.

Virtual networks in a cloud, allow the clients (or tenants) to create a logically isolated subnet, which could be considered a private network. Clients would have the flexibility to choose and assign private IP addresses as they wish in this subnet. This network could be designed to comprise of subnets, and some of these subnets could be private subnets, which are not accessible from outside world (similar to private subnets within a corporate network).

These private networks offer security and isolation and clients have greater control over the security and access policies and can customize the network topologies according to their need.

Advantages of Virtual networks:

1. Assign private IP addresses
2. Restrict internet access to few subnets within the virtual network
3. Customize and apply security policies and access controls for resources within the network
4. Gives a feel of a private datacenter on the cloud.

The virtual private network behaves like a private cloud within the public cloud and we can have connectivity from the corporate network to this private network on the cloud through VPN.

Reference Implementations:

Public Cloud providers like Amazon Webservices, Google Cloud Engine, and Microsoft Azure provide Virtual Private Cloud as a feature.

Cloud Migration

To migrate an existing workload to cloud, multiple options or strategies are available which could be followed as below:

1. Lift and Shift
2. Re-Architect/Refactor to make it cloud native
3. Rewrite

Let us take a look at each of the options below:

1. Lift and Shift:

In this strategy, the application/workload is moved as is to the cloud environment. This is similar to moving virtual machines from on premise to cloud environment. If this approach is used, then the application migrated to the cloud cannot take advantage of any of the cloud features like scalability, availability etc. This approach is generally used as a first step while migrating a large number of small applications or for 3rd party applications whose source code will not be available (and hence cannot be refactored). Sometimes this approach is used just to move away from the traditional datacenter to cloud and then tackle each of the application individually to refactor or re-Architect.

2. Re-Architect/Refactor to make it cloud native:

In this strategy, the application is refactored and often re-Architected to make it cloud native and thus enable the application to be distributed and make full use of cloud features like scalability, availability etc. This approach would be feasible if the application is written using any of the modern languages (this is the bare minimum criteria). Refactoring/re-Architecting cannot be done overnight and sometimes this could involve multiple iterations, so some parts of the application might reside on cloud whereas some might reside on premise, for certain duration, until the entire application is refactored.

3. Rewrite:

In this strategy, the application is rewritten from scratch and deployed on the cloud. This approach is generally used if the legacy application is built on mainframe or if the source code of the application is not available (3rd party application), or sometimes if the cost of refactoring is more than the cost of rewriting.

Serverless Computing

Serverless computing (also sometimes referred to as ‘Function as a Service’) is a new paradigm in computing architecture. Serverless computing is a computing model in which a client submits a request for executing a task. The server lifecycle and the resources are managed by a cloud provider and the client is charged based on certain abstract parameters like memory consumed, CPU cycles or time consumed at sub second level.

Note that serverless doesn’t really mean there is no server. It just means that from a client perspective, the server is a black box, which it need not worry about.

Why do we need Serverless Computing?

Serverless computing eliminates the need for clients to build and manage servers, and also eliminates the need to scale them up or down based on the traffic. Even when there is less traffic, still there is a certain minimum cost involved in keeping the servers running. Thus, Serverless computing provides cost advantage over traditional model.

Limitations of Serverless Computing:

1. Performance Issues:

Serverless Computing might lead to latency issues when the service being provided is not used frequently, since the cloud provider might spawn the server, only when requests are made. Depending on the time required to spawn the server, this might cause undesirable latency issues.

2. Limits on Resources:

Certain cloud providers might apply limit on the resources consumed by the clients and this might impact resource consuming tasks. The cloud provider’s specification needs to be carefully examined before choosing this model.

3. Troubleshooting:

Troubleshooting and debugging could be tedious with serverless computing, since it depends on the diagnostic tools provided by the cloud provider and there may not be a lot of flexibility on troubleshooting options.

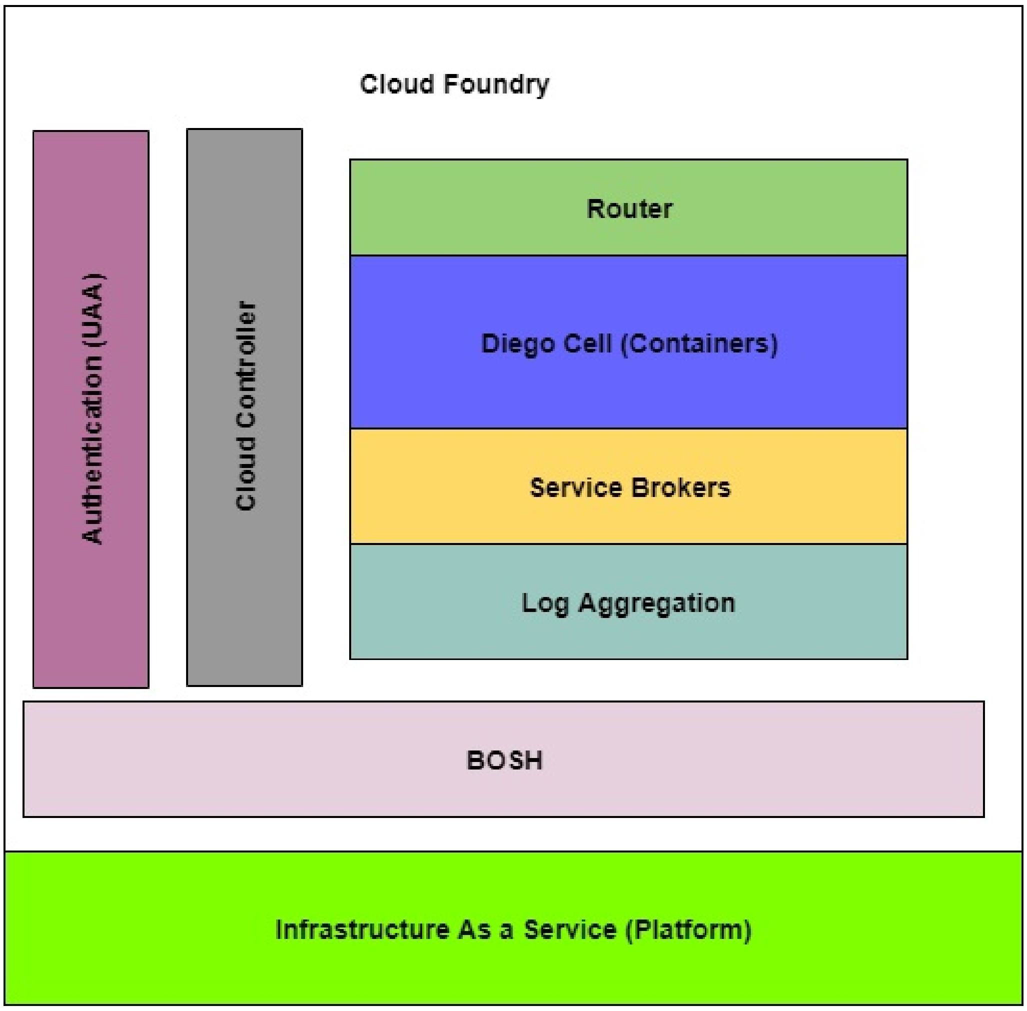
Usecases:

Typical use cases of using Serverless Computing include, running image processing tasks, running social media data analytics etc.

Reference Implementation:

Amazon WebServices offers 'AWS Lambda' which is a serverless computing service.

Cloud Foundry



Cloud Foundry Architecture

Cloud Foundry is an open source platform as a service. There are various commercial variations of Cloud Foundry like Pivotal Cloud Foundry, IBM Blue Mix etc. Cloud Foundry runs on top of an Infrastructure as a service, which should be installed separately. For instance, a customer can install Open Stack and deploy Cloud Foundry on top of it (This is an On-Premise solution).

Cloud Foundry integrates with the underlying 'Infrastructure as a Service' platform to provision resources and a variety of IAAS solutions like Open Stack, VM Ware Vsphere, AWS are supported.

Features of Cloud Foundry:

1. Runtime:

Cloud Foundry provides a run time environment for applications based on the application type. For example, if the application is Java based, it would provide a JVM environment, if the application is Python based, it would provide a Python run time etc. The developers have to just focus their effort on building the application and a single command like ‘push’ would push the application to Cloud Foundry, setup the run time environment and get it up and running. Also, there is no hard tie up to any Cloud Foundry API within the application. The same application, which would be built for any other environment could be used for Cloud Foundry as well.

2. Marketplace Services:

Cloud Foundry provides market place services like, databases, Message queues etc., as managed applications within Cloud Foundry. For instance, if an application has to connect to a database, we just have to bind the application to the database service which is offered through marketplace. The services could be hosted services on Cloud Foundry platform and managed by Cloud Foundry or it could be services which are provided by third party vendors.

3. Logging, Scaling:

Cloud Foundry also provides a lot of other features, like scaling (based on memory or CPU), logging features etc.

Thus with Cloud Foundry, we can get an application from cradle to production very quickly, without the hassles of environment setup, infrastructure setup etc.

Components of Cloud Foundry:

Following are some of the components of Cloud Foundry:

1. Router:

Every application which is deployed to Cloud Foundry will be mapped to a hostname (IP address and port). A router (known as 'Go Router') component redirects incoming traffic to the application and takes care of load balancing across multiple instances of the application.

2. Diego Cell:

Cloud Foundry creates containers (known as 'Diego Cells') internally and deploys applications on these containers.

3. UAA:

Cloud Foundry uses a component knowns as 'UAA' for authentication and authorization. This module helps in authenticating and authorizing users to Cloud Foundry Platform itself.

4. Service Brokers:

Cloud Foundry uses 'Service Brokers' to make services available in the marketplace. Each service broker would take care of provisioning required resources for each service and initialize necessary credentials. Attaching a service to an application is achieved through a process known as 'Binding'. For example, a service broker for MySql database, would create a new database and create a new user for that database. Thus when a MySql service is binded to an application, the service broker would provision the necessary resources for use.

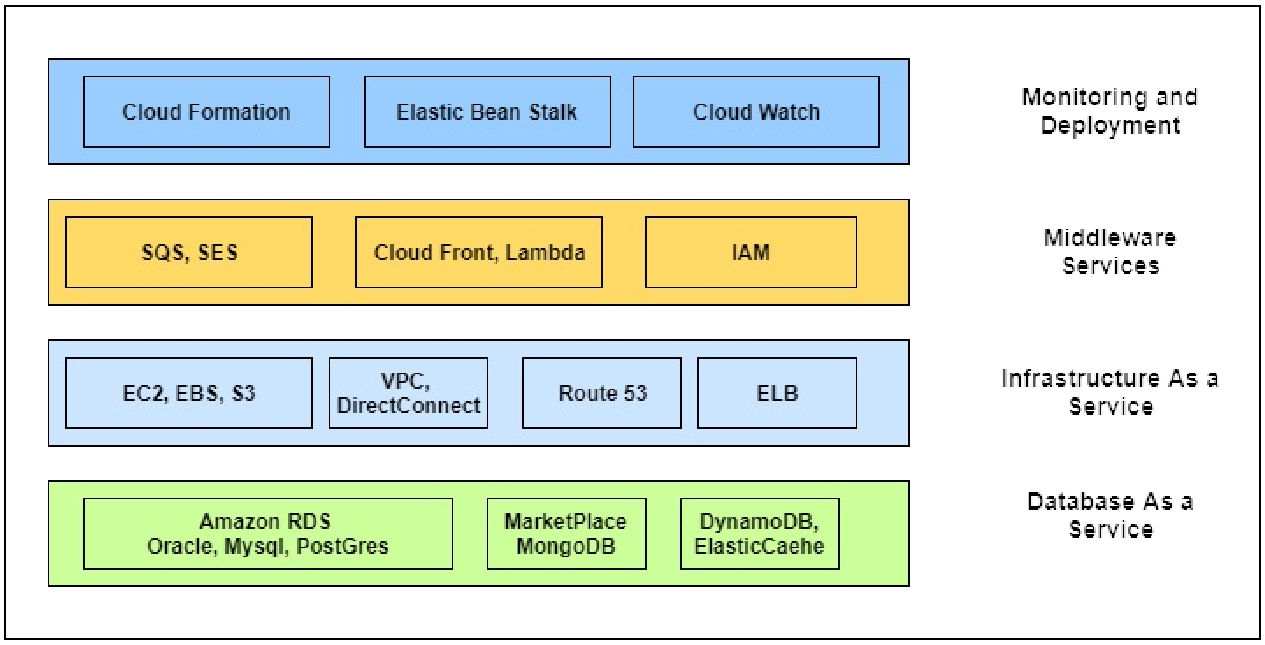
5. Cloud Controller:

'Cloud Controller' component is the most crucial component of Cloud Foundry and it will take care of managing the other components of Cloud Foundry and also performs regular health checks of ther other components. Most of the communication to the various interfaces exposed by Cloud Foundry platform happens via the 'Cloud Controller'.

6. BOSH:

The BOSH layer takes care of release management of the Cloud Platform components and it is used in deploying the platform itself. Also, BOSH is used in interacting with the underlying IAAS layer to provision infrastructure.

Amazon Web Services (AWS)



AWS Components

In this section, we will take a look at Amazon Web Services (AWS) platform and the various features provided by it.

Following are a few products and services offered by Amazon Web Services, and this should give an idea of the capabilities provided by the AWS platform.

Infrastructure As a Service:

The IAAS components include compute capacity, storage, and network components.

1. EC2 instance:

EC2 instances provide compute capacity with different combinations of CPU and Memory and obviously varying price tags, which could be selected based on requirements. When we create an EC2 instance, AWS will reserve the specified amount of CPU and memory for us. An EC2 instance is analogous to a virtual machine. An EC2 instance can be created from a predefined image which is known as 'Amazon Machine Image' (AMI). Note that an EC2 instance is ephemeral, meaning, the content of the EC2 instance is volatile and could be lost at any time. EC2 instance does not guarantee persistence. However, AWS provides an SLA such that, if we provision N number of EC2 instances, then AWS will guarantee that number with a high availability (refer to AWS docs for how much nines they offer).

2. Elastic Block storage (EBS):

EBS or Elastic Block storage is a storage component for for EC2 instances and it could be attached to any EC2 instance. It will act like a volume storage and will provide persistence to the EC2 instance.

3. Elastic Load Balancing (ELB):

ELB is a scalable Load balancer solution which provides load balancer capabilities, but in addition, can scale horizontally based on traffic. This is one of the unique offerings and can guarantee high performance in high volume applications. A traditional load balancer cannot scale horizontally and we would need to estimate and install a fixed number of active load balancers. However, if the traffic grows or shrinks, we cannot add or remove load balancers on the fly in a traditional model. An ELB, can add or remove load balancers on the fly and will give us cost edge (since when traffic is low, the ELB would scale down) and also provide better performance.

4. Amazon VPC:

A virtual private cloud, which provides an isolated network with subnets, which could be either internet facing or private subnets. (VPCs in general, were covered in earlier sections).

5. Route 53:

Route 53 is a DNS as a service where domain names could be configured. It is similar to a traditional DNS server, but this an offering on the cloud from Amazon.

6. EC2 Container Service:

ECS is a Docker based Container Orchestration service. (Container Orchestration in general, were covered in earlier sections).

Database As a Service:

AWS provides quite a wide variety of databases as a service, including RDBMS and NoSql databases like DynamoDB, Elastic Cache. AWS also provides a marketplace, wherein third party vendors can provide database as a service offering. For example, MongoDB is provided as a service by MongoLab. The SLA of that database availability , maintenance etc. are specified by this vendor.

Relational Database Service (RDS):

Some of the popular RDBMS, like Oracle, MySql, PostGres etc. are provided as a service through RDS. An RDS instance could also be provisioned within a specific VPC (in which case, the data resides within the VPC).

Middleware As a Service:

AWS provides quite a lot middleware service like Authentication services (IAM), messaging queues (SQS), email services (SES).

Monitoring and Deployment:

AWS provides services to monitor applications/infrastructure like 'Cloud Watch' and also frameworks like 'Elastic Bean Stalk' which is a platform as a service which helps in quick deployment of applications.

Application Architecture

This section explains the various Application Architecture concepts in detail.

REST

REST stands for Representative State Transfer. It is an architectural style for communicating between two entities. Restful architecture is resource oriented. The services expose a set of resources to the client and the client can perform a set of operations on the resources which modify the state of the resource. It is a stateless protocol.

Why is REST important in the Cloud?

REST is one of the most common, standard and the simplest way of communicating between two entities or resources. REST is supported by every major programming language, used on both client side and server side. REST is used/supported by mobile applications, browsers, and backend systems. Many modern databases also provide a REST wrapper around their data APIs. Most of the Web 2.0 revolution happened around REST. So naturally, REST finds an important place in the Cloud design and architecture.

RESTful Design

Restful service design requires a paradigm shift in the way we think about designing services. In the SOAP world, it was all about operations. So we used to think, what will this web service "DO". But that is something which we should not do while designing Rest services. Let's take for example an order entry application. In the SOAP world, we would think about services like createOrder, updateOrder, addLineItems etc. To design the same services in Restful world, we need to identify all the resources in the application which needs to be exposed. These resources need not necessarily have a one to one mapping to the underlying schema. It is just the resources which matter to the client.

In our example, the resources could be Order, LineItem, Customer etc. We also need to establish a hierarchy amongst these resources for e.g., Customer -> Order -> LineItem.

How to identify the resources in REST?

In REST, the resources are identified by using URI. The URI should be something like a directory structure listing so that the client will not have too much of a trouble to guess the URI. The URI could also embed identifiers to pin point to a particular resource. For e.g., the url can embed an order ID, a customer ID etc.

In our example, the URI to identify an order with an order ID of 1000 for a customer Paul, could be something like: /customers/customer/paul/orders/order/1000/ Note that we could easily guess the hierarchy and drill down to the particular order. Removing /order/1000 will return an URI: /customers/customer/paul/orders/

The above URI will identify all the orders ordered by paul.

How to operate on Resources?

In order for the client to operate on the resources, Rest relies on HTTP protocol and uses the standard GET, PUT, POST, DELETE operations. The following table explains each of these operations.

HTTP Method	Description	Example
GET	Retrieve a resource	Retrieves the order with order ID 1000
PUT	Create/update a resource (completely replace existing resource). Used when we know the URI which will be created/updated	Update an order with ID 1000. The order details could be in Request body.
POST	Create a resource when we do not know the URI which will be created	Create an order with details. The order details could be in Request body.
DELETE	Delete a resource.	Delete an order with ID 1000.
HEAD	Identical to GET except that the server MUST NOT return a message-body in the response. HEAD should be used to return meta data about the resource returned by GET. Clients can cache the results of HEAD.	HEAD could be used to check if a resource exists or to check the last modified date.
PATCH	Updates a resource partially. This is generally used when we want to update only a few attributes of a resources and not the entire resource. The client can send only those attributes of the resource which needs to be updated.	Update order, with ID 1000 to update Status to Pending.

PUT vs POST:

There is an argument over usage of PUT vs POST as to when we should use PUT and when we should use POST. PUT requires the operation to be idempotent, while POST does not. i.e., multiple invokes to PUT the same resource should result in the same outcome.

Let us take a closer look at this: The URI in consideration is:
/customers/customer/paul/orders/order/1000

A PUT request to the above URI, should create a new order with an ID of 1000. If an order already exists with ID of 1000, the resource should get replaced with the data provided in this request. So, no matter how many times we invoke PUT for ID of 1000, the outcome will always be an order with ID 1000. That is idempotency. The definition of POST says that, POST should be used to create a child resource as well as the URI. i.e., the client is not aware of the URI being generated.

1. Use case for POST:

A typical use case for POST is, when a client intends to "Create an order and get back an ID which could be used in further requests". So, a POST request to '/customers/customer/paul/orders/order' will generate an Order with ID (say) 1000 and it will also create an URI '/customers/customer/paul/orders/order/1000'.

Subsequent PUT requests to the above URI will update the order with ID 1000. So, POST could be used to create orders when the server will generate unique IDs rather than client specifying the IDs.

2. Use case for PUT:

A typical use case for PUT is, when a client intends to "Create an order with ID 1000 which could be used for further requests". Note that here, 1000 need not be the primary key on the server side. That detail is hidden from the client. The server might generate its own primary Id and correlate it to 1000. All that the client cares is that, it should be able to get/update using that ID.

How to handle errors?

Exception scenarios could be handled by returning HTTP error codes like 400,404, 403 etc. These are standard HTTP codes and the standard should be followed depending on the scenario. For example, a HTTP code of 404 means that the URI doesn't exist.

What are the Response formats?

RESTful services can return XML, JSON etc. which is driven by the HTTP accept header. The service needs to explicitly set HTTP header indicating the response MIME type which it is returning. The client would then need to set the HTTP accept header accordingly. For example, a HTTP accept header of 'application/json' indicates that the client is ready to accept JSON response.

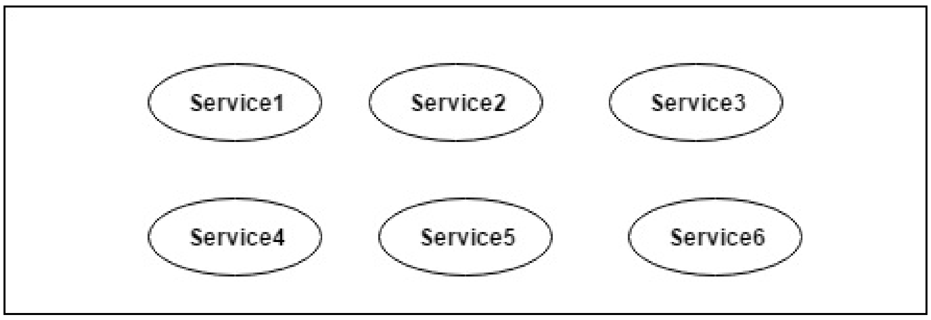
HATEOAS

An interesting point to note is that, in REST, we can also return URIs in the response. Let's say after a POST request to create a new order, instead of returning an ID, we can return the new URI, which could be used by the client for further GET/PUT requests. This can be done by returning the URI in the location header. For e.g., the service can return a header “Location: /order/1000”. This means, the client can issue GET requests at /order/1000 to retrieve the newly created order. This concept is known as Hypermedia as the engine of application state (HATEOAS).

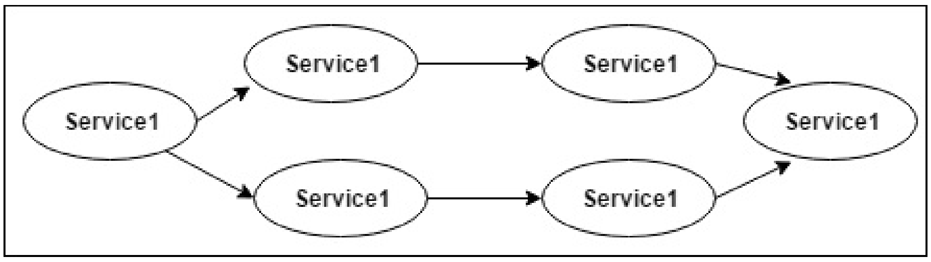
How to describe the service API?

Similar to WSDL (Web Services Description Language), in the Rest world, we can describe the service API using WADL (Web Application Description Language). It is an xml based representation of the service API.

Service Oriented Architecture (SOA)



Loosely coupled Services



Service Orchestration

Service Oriented Architecture

Service Oriented Architecture (popularly known as 'SOA') is an architectural style of designing systems using loosely coupled services which communicate with each other using well established protocols and message payloads.

A service is a discrete unit of functionality that can be accessed remotely, acted upon and updated independently.

SOA principles have evolved over time and away from applications having tightly coupled and rigid communication mechanisms which are hard to integrate with other systems. SOA aims at building well defined independent services which could be built using heterogeneous technology stack but conform to principles of SOA. These set of services could be easily integrated and each of these services could evolve independently of others.

Guiding principles of SOA:

Following are a few guiding principles of SOA (From Thomas Erl's, 'SOA Principles of Service Design'):

1. Service Abstraction

The service should abstract out the underlying complexities of implementation (functionally and technically).

2. Service Contract:

The service should adhere to a well-defined contract which is exposed to the clients of this service. For e.g., request response messaging formats and the contract of what the service offers.

3. Service Autonomy:

Services should control the underlying functionality and evolve independently without affecting other services.

4. Service Granularity:

Services should be designed to be sufficiently granular.

5. Service Composability:

Services could be used to compose multiple services into a bigger service which is well orchestrated.

6. Service Discovery:

Services should be self-discoverable.

7. Service Loose coupling:

Services should be loosely coupled and should use open standards for communicating amongst themselves. Loose coupling of services allow for independent evolution of services.

8. Service reusability:

Services should be designed by keeping reusability in mind. A service should offer specific functions with a well-defined context and the service could be potentially reused across the enterprise.

SOA Implementation:

The principles of SOA are independent of the implementation technology and there are lots of technologies which enable SOA adoption.

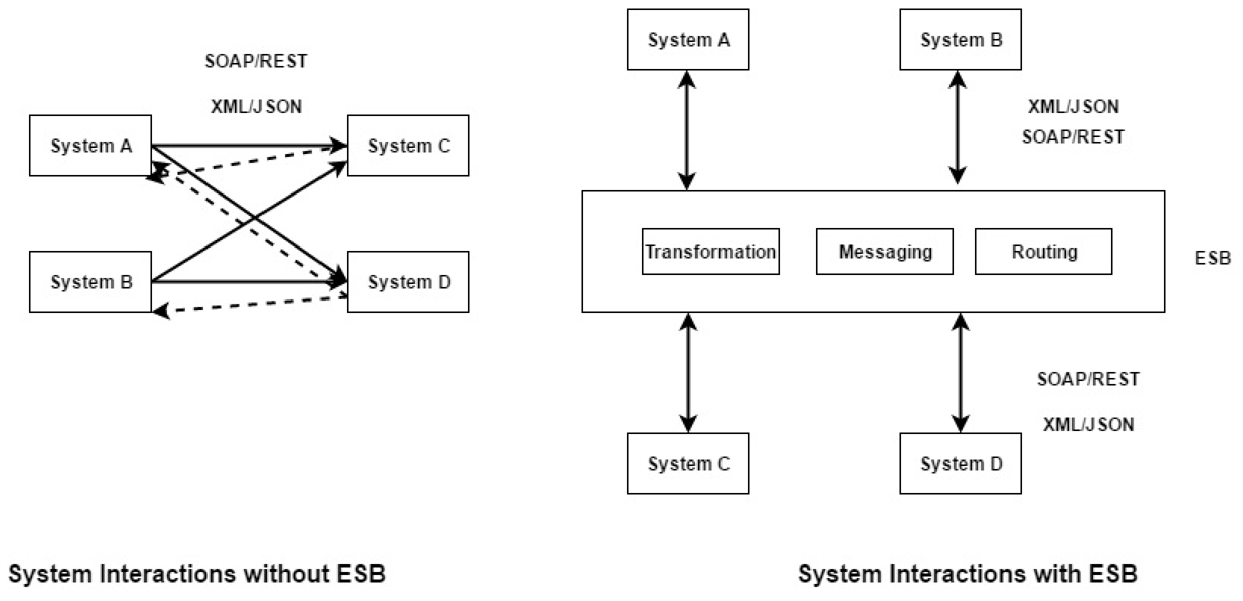
For messaging protocols, we can use SOAP or REST and Synchronous or/and Asynchronous modes of communication could be used.

For service orchestration, technologies like BPEL (Business Process Execution Language) which provides a language context for composing services could be used. There are a lot of commercial and open source vendors who support BPEL.

Reference Implementations:

SOA tools like Mule ESB, Oracle ESB, WSO2 etc. provide SOA implementations with features like service bus, message transformation, routing etc.

Enterprise Service Bus



Enterprise Service Bus

Enterprise Service Bus (ESB in short) is an architectural pattern which is used to integrate heterogeneous systems by using service-oriented architecture. The systems could be different in various aspects as listed below:

1. Built on different technology stacks, like Java, .Net, PHP etc.
2. Support different communication protocols like SOAP, REST and FTP etc.
3. Support different message formats like XML, JSON etc.
4. Support different mode of communication like synchronous, asynchronous.

An ESB is a centralized hub where we can integrate various service end points, implement message transformations, apply security rules etc.

Benefits:

1. Loose coupling of systems and ease of integration irrespective of how the systems are built. Let’s say we have systems A, B, C, D which are all built using different technology stack and which support different message formats. Now, if each of these systems has to communicate with each other, we would need 6 end points in total, if we establish point to point communication.

Also, this approach would pose the following challenges:

- Implement message transformations in every individual system
- Every system should know the location of every other system
- Every system has to deal with interoperability issues
- Every system may have to implement different security mechanisms, matching that of the destination system

If we use an ESB (as in the above diagram) and connect all these systems to the ESB, then we can address all the above challenges. Each system needs to communicate only with the ESB and be ignorant of the destination system’s location, message formats etc. The message format transformation/security can be implemented on the ESB. The advantage of this approach is that we can add more systems to the mix, without much effort. Also any change to the service interface could be addressed at the ESB layer.

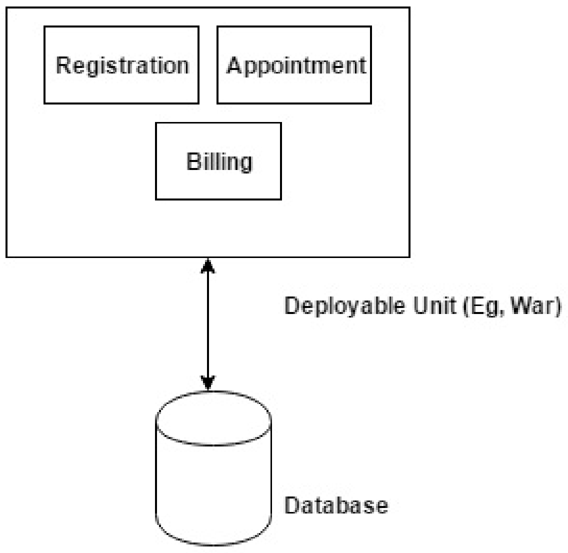
Challenges:

1. Use of a centralized hub like ESB acts as a single point of failure and might bring down the entire eco system. The ESB scalability and its support for distributed computing are very crucial and will have a major impact on the overall scalability of the entire system.
2. Development on ESB requires specific skills like XML, XSLT, and XQUERY and is generally managed by a single team. This might act like a bottleneck, especially if several teams are dependent on one single team which works on ESB. Differing project priorities might cause potential clashes due lack of bandwidth. This is more of an organizational challenge.

Reference Implementations:

Mule ESB is a very popular service bus; similarly other vendors like Oracle, IBM have their own service bus implementations.

Monolith Architecture:



Monolithic Architecture

In this section, we will take a look at one of the traditional architectures, which is referred to as monolithic architecture.

Consider a hospital management solution. It might have multiple modules like Patient Registration, Doctor Appointment System, Room booking System, Billing System etc.

In a traditional architecture, these modules might be deployed as a single artifact (for example Jar/WAR in Java) and they connect to a single database, as shown in above diagram.

As the hospital expands its business, the number of patients' increases multifold and the application will need to scale accordingly. However, scaling might not be that easy with this architecture. We cannot scale individual modules and we need to assign memory/CPU to the entire application leading to inefficient resource usage. For example, on a day where number of patients getting discharged is high, we might need to scale the billing module alone. This is one of the drawbacks of monolithic architecture, which will not fit into a distributed ecosystem.

Alternative:

An alternative to Monolithic architecture is a Microservices Architecture, which is explained in later sections.

Cloud Native Applications

Cloud Native Applications are a class of applications which are architected to fully make take advantage of a cloud’s capabilities like Scalability etc.

Why do we need Cloud Native applications?

Before taking a look at cloud native applications, let us assume that a traditionally architected application has been deployed on a cloud infrastructure. Since the infrastructure is provided by cloud, the infrastructure is smart. i.e., the infrastructure is elastic, (it can scale up or down), the infrastructure is resilient (if a virtual machine goes down, the cloud platform would guarantee that it is brought back online). However, the question is, whether the application is smart enough to run on cloud?

For an application to effectively run on the cloud, we can pose the following questions to ourselves:

1. Can we scale the application horizontally? If we have multiple instances of the application, will it continue to function normally?
2. Can we scale independent modules of the application?
3. Is the application writing anything to local file system? If yes, then if we have multiple instances of the application, then it will create issues, since changes made to the local file system, remain local to that instance and will cause inconsistency. Also, if an instance goes down, the file system changes are lost.
4. Is the application maintaining session locally? If yes, then if we have multiple instances of the application, then it will create issues, since the local session will not be available to other instances and if an instance goes down, the session information is lost.
5. Is the application resilient to instances going down? The cloud infrastructure will bring up the instance very quickly; however, what will be the impact on the applications?

There are a few aspects or attributes which are required for an application to effectively utilize all the advantages or features that a cloud platform provides. An application which does this is known as 'Cloud Native'.

Following are a few features of Cloud native applications. Notice that, most of these correspond and answer to the questions posted above:

Characteristics of a cloud native application:

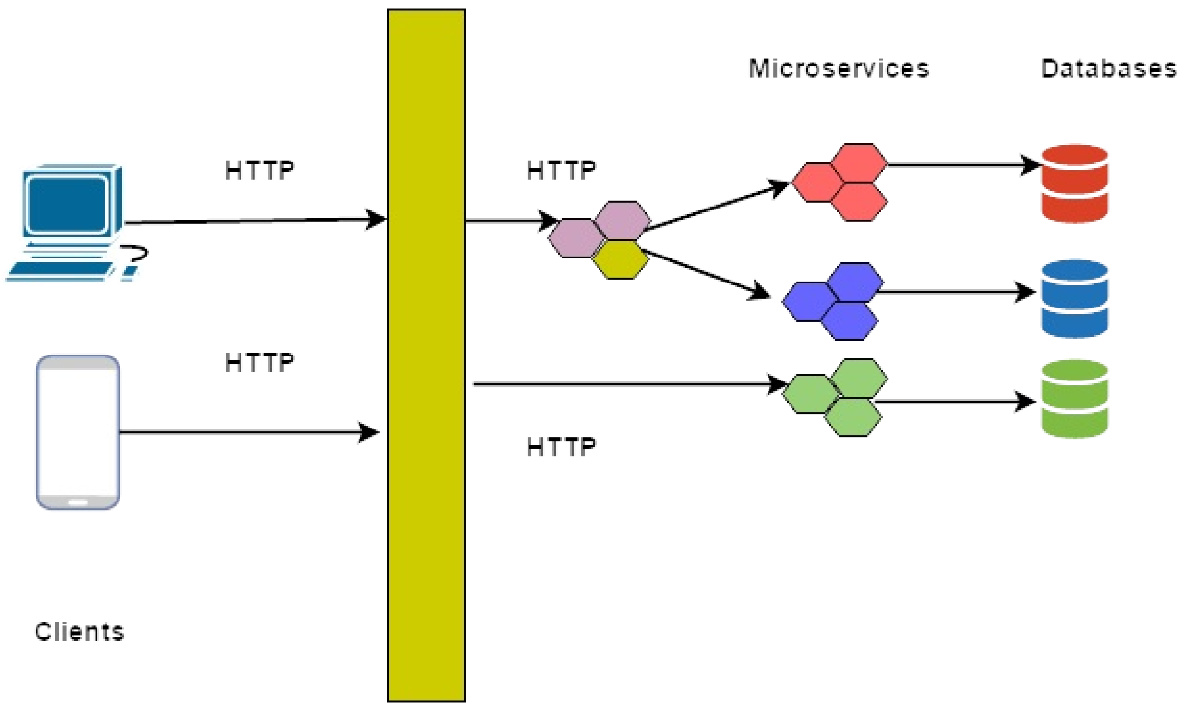
Following are the characteristics of a cloud native application:

1. Could be scaled horizontally, without affecting the functional behavior of the application.
2. Allows independent modules of the application to be scaled, thereby making best utilization of the resources.
3. Does not write to a local file system. If there is any such use case, it would write to an object storage system, which is distributed in nature and could be accessed by all instances.
4. Does not maintain a local session. Instead, it uses a distributed cache to persist session, so that, all instances could access the same session data.
5. Is resilient to instances going down and will be as stateless as possible, so that new instances could be added or existing instances removed without affecting any functionality.

12 factor Applications:

A set of principles known as '12 factor application principles' is popularly used in designing Cloud Native applications. 12 factor application principles could be found here: 12 factor

MicroServices



Microservices Architecture

Let us take a look at a popular architectural pattern known as ‘Microservices’.

The Definition of Microservices by Adrian Cockcroft is as below: "Loosely coupled service oriented architecture with bounded contexts".

Microservice is an architectural pattern wherein we develop small, manageable services with a well-defined API and which are independently deployable. Each of the Microservices typically will be backed by its own database. So in essence, every microservice is an application in its own. This is in contrast to the traditional monolithic applications. Monolith, as the name implies is a single huge application, which has tightly coupled modules. Traditionally we used to build applications as a monolith, i.e., the entire application is packaged as one single deployable unit. For instance in the J2EE world, the deployable unit used to be a large EAR/WAR file and one single large database.

Why do need microservices?

1. Roll out features quickly to production:

The customer might need minor changes to functionality which should be rolled out at the earliest, as they have a higher business impact. Microservices enable us to make small iterative changes, test and deploy those changes quickly to production.

2. Ease of maintaining and adding functionality:

With loosely coupled microservice architecture, each service is loosely coupled to other services and each service would have a well-defined boundary (which is known as bounded context). This allows for each service to independently evolve and it would be easier to add new features to individual microservices. This makes the overall system easier to maintain from a functional standpoint. In a traditional monolithic application, there could be a lot of tight coupling amongst modules and adding new features could sometimes be nightmarish.

3. Scale out the application based on modules:

Very often we observe that few modules of our application face a huge load compared to others. We might want to dedicate more resources to these modules instead of scaling out the entire application. However, in a monolithic application, we may not be able to do this at a module level (since we have a single huge deployable unit), thus ending up wasting money/resources by scaling the entire application. However, with microservices, we can scale individual services easily, thereby saving compute/memory resources.

4. Flexibility to choose the right database, also known as Polyglot Persistence:

With microservices architecture, we can choose the right database flavor for the right service. In a monolith, generally, we will be forced to retrofit our design to fit the database which is already chosen for the application. However, in a microservice, we have the liberty to select the right database which fits the requirement. For instance, in an Ecommerce application, a microservice which provides Product Reviews as a service, might chose a NoSQL database like Mongo DB to store the reviews.

5. Flexibility to choose the right programming language, also known as Polyglot Programming:

With microservices architecture, we can choose the right programming language which fits the requirement. One service could be written in Ruby, while other in Python and so on.

Reference Implementation:

An Ecommerce application could be composed of multiple microservices as below:

1. Service for Product Catalog
2. Service for Customer Profile
3. Service for Product Reviews
4. Service for Shopping Cart
5. Service for Order Fulfillment

If each of these services is exposed through a well-defined API, then changes to the individual services could be made quickly and rolled out quickly (as long as they don't change the API interface).

Challenges posed by Microservices Architecture:

Let us take a look at the challenges which we would face while adopting Microservices Architecture. Not everything is simple with Microservices. It comes with a baggage of its own problems.

1. Service Discovery
2. Logging
3. Monitoring
4. Security
5. Performance
6. Deployment challenges
7. Database challenges
8. API Management
9. Integration Testing
10. Fault Handling

Microservices Challenges

Microservices architecture, though it sounds interesting and proving to be hugely popular, comes with its own set of challenges. We will look at a few challenges posed by this architecture.

A typical Microservices based application deployed in the Cloud could have a large number of services and each service in turn can have a large number of instances (scaled horizontally). This setup can pose varying challenges as detailed out below:

1. Service Discovery

With a large number of services/instances, it becomes important to identify and locate each service/instance. Similarly, as we add/remove new services/instances, other services need to know about this.

Available solutions:

Use a service registry which maintains a registry of services, their end points and their instances. Some load balancers also maintain service registry. We can use client side/server side load balancers which also maintain service registry. E.g., we can use Load balancers like Eureka, Consul.

2. Logging

With a large number of services/instances, it becomes important to collect the logs at a centralized location. If the logs are not centralized then, we would have to go into different servers to check the logs for troubleshooting and this process is very cumbersome and painful. If a request spans across a number of services, it is also important to trace the request across all services.

Available solutions:

- Centralized Logging solutions like Splunk or Elastic Search, Logstash, Kibana (Known as ELK Stack), or cloud based log aggregators like Papertrail could be used to collect the logs at a centralized location.
- Logs could be enhanced by adding suitable header info (meta data info) which identifies a request and ties the log message to a request.

3. Monitoring

With a large number of services/instances, it becomes very crucial to monitor the service instances, monitor their performance, health, memory status etc.

Available solutions:

Monitoring tools like New relic, Sensu could be used to monitor the services, gather performance metrics and configure alerts.

4. Security

Because of the distributed nature of the application, securing every end point/services takes high priority. Also, a single insecure end point can prove to be a weak link in the chain and lead to catastrophes.

Available solutions:

The end points could be secured using security protocols like OAUTH.

5. Performance

Since a single cohesive monolithic application is now decomposed into numerous independently deployable microservices, it might cause a dip in performance and will need sufficient performance monitoring and tuning as needed.

6. Deployment challenges

With a large number of services/instances, deploying each service can become challenging. This might become complicated if different services are built using different programming languages and require different run time environments. Also, deploying to a huge number of servers could pose challenges with rollback if something goes wrong.

Available solutions:

- Use automated deployment processes
- Use Platform as a Service
- Use Containers for deployment
- Use Blue Green or Canary deployment techniques

7. Database challenges

Use of polyglot persistence might require diverse database skills (SQL/NOSQL), while administering the application. Microservices generally use one database per service and the database changes and application deployments need to be synchronized.

8. API Management

With microservices architecture, generally API driven development is followed and it becomes important to properly manage the APIs. The management of APIs would include the following aspects:

- Versioning of APIs
- Maintaining backward compatibility of APIs
- Documenting API interfaces
- Metering of APIs (If the APIs are exposed to third party vendors or to the public and if each invocation of the API is billed, then we would need to log and price each API invocation).

Available solutions:

A lot of API management solutions like Apigee, WSO2, and Kong are available which could be used.

9. Integration Testing

With microservices architecture having large number of services, inter communication between services becomes inevitable. This brings in Integration testing challenges with each release.

Available solutions:

Use Integration testing tools like Citrus or Pact.

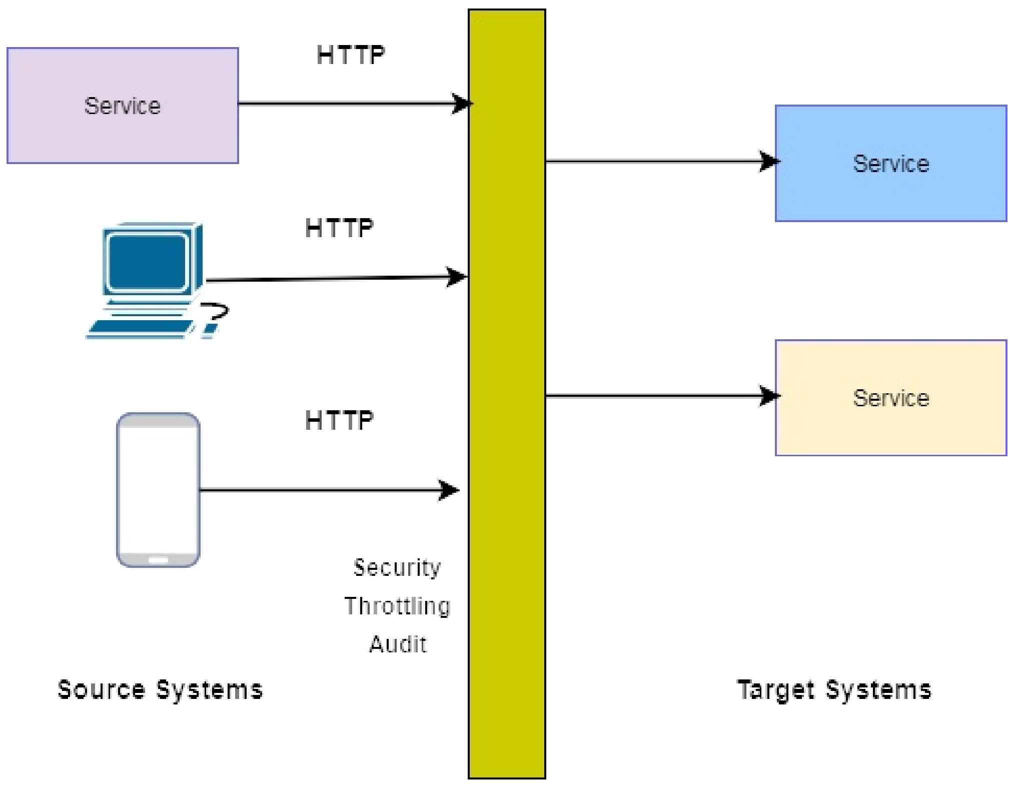
10. Fault Handling

With microservices architecture having large number of services, a failure in one service could lead to a cascading failure in all dependent services and this could quickly gulp system resources. Obviously, robust fault handling is very crucial and needs to be designed and built into the system.

Available solutions:

Use patterns like Circuit breaker which works like electric circuit breakers. If a service call fails, it will fall back to a Fail over implementation, until the service comes back to normalcy. Also the system should fail fast, so that the issues are quickly noticed and resolved.

API Gateway



API Gateway

API Gateway is a pattern commonly used in Microservices Architecture. API Gateway serves as an Edge Server in the overall system architecture.

Why do we need an API Gateway?

1. Different clients (like mobile devices, tablets, browsers etc.), might need and demand services with different granularity. For instance, a particular client might need a service which returns a heavier payload and doesn't want too many calls, back and forth to the server whereas; another client might need a smaller payload and might require additional calls to the server to get more data as needed.
2. Exposing all the individual service end points to the client may not be a good idea. Exposing one end point would be easier to manage from a client perspective.
3. Managing and applying security is easier if one common end point is exposed.
4. In a cloud environment with a pay per use model for services, metering the service usage is easier if it is centralized.

An API Gateway could help address the above concerns and provide services like routing, transformation and service aggregation, in addition to applying security. An API Gateway can also manage versions of service APIs which are exposed to clients.

Rate Limiting and Throttling:

An API Gateway implements throttling of requests and also applies rate limiting so that the number of requests could be monitored and controlled.

Rate Limiting:

When an API is exposed to consumers, there is a possibility of the API being abused intentionally or inadvertently, in the absence of controls. Rate Limiting will limit the number of times an API could be invoked by a consumer in a specific window of time. For example, the API could be invoked let's say 200 times in 4 hours. The charge back to the consumer could be applied accordingly. If the consumer exceeds this limit in a given window, then the API gateway would reject the request and send back appropriate error (Generally, a 403 status code will be returned).

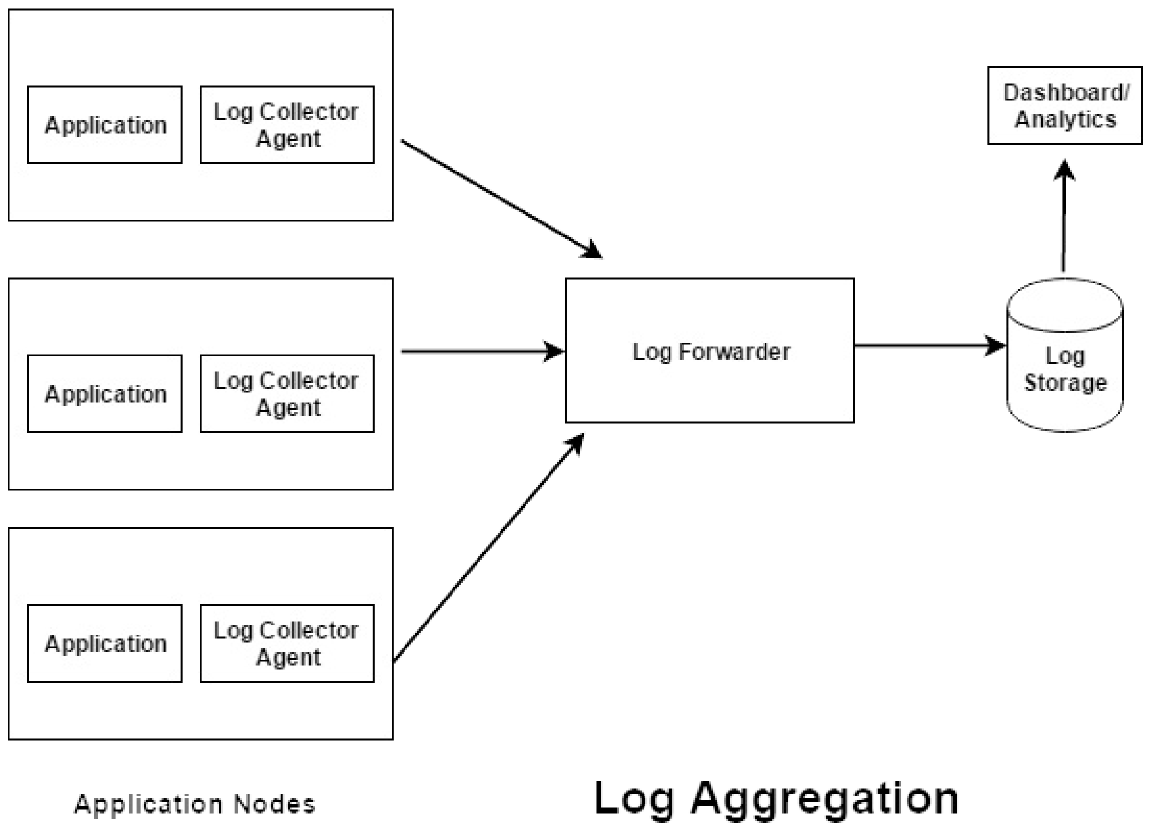
Throttling:

Throttling is a process of queuing the requests, when the rate limit has been applied. i.e, when the number of requests have exceeded in a given window, the future requests could also be throttled, i.e, instead of returning 403, the request would be queued for a specified amount of time, say 100 ms. After this time has elapsed, the request would be picked up again for processing and if the window has still not elapsed and the quota is exceeded, then 403 error is returned back to the client.

Reference Implementations

A few libraries/solutions which implement API Gateway are Netflix Zuul, WSO2, Kong, Apigee.

Log Aggregation



In a cloud environment employing a microservices architecture, one of the major challenges is to collect the logs from different services (and multiple instances of a service) and virtual machines for purposes of troubleshooting and analysis.

One of the solutions is to use a log aggregator. A log aggregator system is a system, which collects log data from different systems/services/virtual machines at a centralized location and provides a uniform way of accessing and analyzing logs.

Log Aggregator Sub-systems:

A log aggregator generally has 3 sub-systems as listed out below.

1. Log collector
2. Log forwarder
3. Log storage system
4. Visualizer and a Dashboard

Each of these sub-systems which are illustrated in the above diagram is explained below.

1. Log collector

A log collector is an agent which will be deployed on the individual servers and is responsible for collecting logs from the application and forwarding it to a common server.

2. Log forwarder

A log forwarder is a component server which will accept logs from the collector agents; it might perform formatting of logs as needed and would communicate with the storage system to store the logs or it might also integrate with an alerting system to trigger alerts accordingly.

3. Log storage system

A log storage system is a database solution which would store the logs and provide search capabilities.

4. Visualizer and a Dashboard

A visualizer provides a visual representation of the logs in a dashboard format by retrieving it from the storage system and would also support running analytical queries on the logs.

Reference Implementation:

A variety of technologies and frameworks, open source or commercial are available which provide log aggregation.

One such technology stack is ELK stack (or another variant called as EFK stack). ELK stands for Elastic Search, Log Stash and Kibana (The F in EFK stands for Fluentd).

1. Log collection and forwarding could be done by Log stash or by FLuentd libraries
2. Log storage could be performed using Elastic Search.
3. Visualizer and Dashboard could be served using Kibana

Most of the cloud platforms (Platform as a Service), support Log aggregation by channeling all the logs from the platform to a common point and support various log aggregator integrations.

Aysnc Processing

In this section, we will take a look at Async processing, which is a way of processing, where-in the actual task to be executed is deferred to, a later point in time.

Why do we need Async processing?

If the task to be executed is a computationally intensive operation and is going to take significant amount of time to complete, we may not want the thread which initiates this task to wait till the task executes. For e.g., Let us say we have a user Interface, which initiates file processing task by uploading it to the server. The file processing could take say a few minutes. It may not be a good idea to keep the User Interface in a waiting mode until the server completes its task.

We can make use of Async processing to submit the task and have the user interface return back to receive user actions. Once the task is completed, the user could be notified about its status.

Aysnc processing can be performed on the server side as well as on the client side, depending on the requirement.

How to achieve Server side Async Processing?

There are various ways to achieve Async processing:

1. Using Native threads
2. Using messaging systems like Java Messenger Service (JMS), Kafka, specialized queues like Rabbit MQ, Active MQ

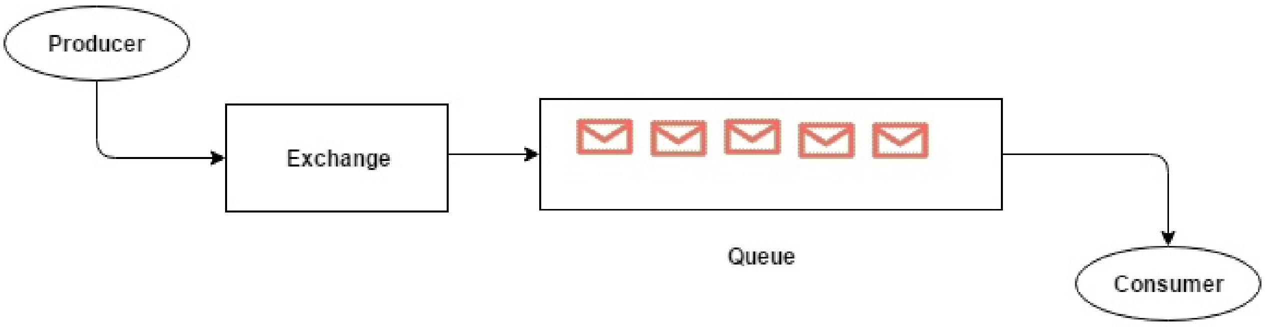
How to achieve Client side Async Processing?

Client side Async processing techniques like Websockets or Promises feature available in UI frameworks like Angular JS, AJAX provide similar async capabilities.

Queues and Topics

In this section, we will take a look at Queues and Topics.

Queue:

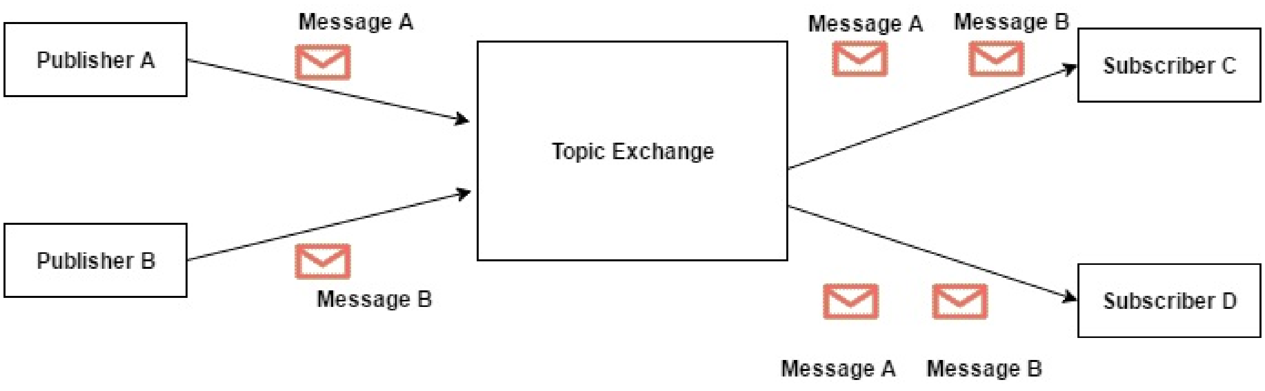


Queue System

In the context of messaging, a Queue allows point to point communication between producers and consumers in an asynchronous manner. Following is an execution path for a Queue:

1. The producer publishes the message to an exchange
2. The exchange would place the message on a Queue
3. A consumer which has subscribed to read from the queue would read the message. Once the consumer reads the message from the queue, the message will be removed from the queue.
4. The consumer would acknowledge the receipt of the message.

Topic:



Topic System

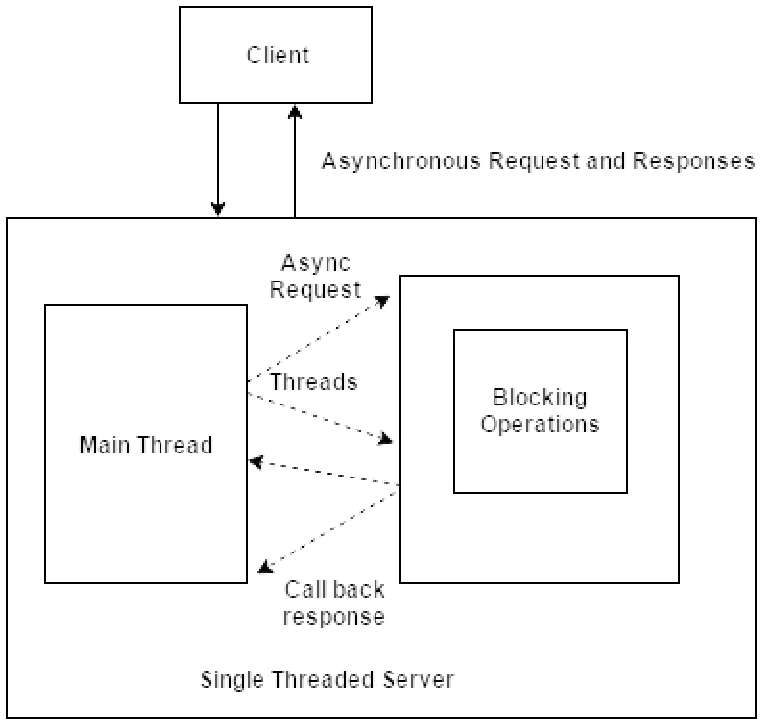
In the context of messaging, a Topic allows for Publish/Subscribe model of communication, where a producer would place the message on a Topic. One or more consumers can subscribe to the topic and the message would be broadcasted to all the consumers who have subscribed to that topic. In the above diagram, we can see that publishers A and B place messages A and B respectively on the topic and the topic broadcasts both of these messages to each of the subscribers (C and D) respectively.

Reference Implementations:

A lot of commercial and open source vendors provide Queue/Topic implementations like Rabbit MQ, Active MQ, TIBCO.

Single Threaded Event Loop Model

In this section, we will take a look at popular asynchronous model known as 'Single Threaded Event Loop Model' which is widely used in building scalable server side applications.



Single Threaded Event Loop model

Traditionally, web servers which serve HTTP requests are multi-threaded. They spawn a new thread for every HTTP request which arrives. This thread would be responsible for sending the response, which might include database calls which are Blocking I/O operations. This kind of server may not scale as the request traffic increases exponentially.

A new model, known as 'Single Threaded Event Loop Model' has emerged wherein instead of spawning one thread per request; a single thread would cater to all requests. But how does this scale?

This single thread would perform the processing and whenever it has to make blocking I/O operation, it would spawn a separate thread to perform that operation. Upon completion, the thread which performs the I/O would make a call back to the main thread the response would be returned to the client. As shown in the above diagram, all blocking operations are delegated to a separate thread asynchronously and the once the blocking operation completes, it would perform a call back and provide the response, which is eventually returned to the client.

Reference Implementations:

Frameworks like Node JS, Reactive Java support Single Threaded Event Loop Model.

Kafka

In this section, we will take a look at Kafka, which is a distributed streaming platform.

Kafka is a distributed messaging system which uses Publish/Subscribe model of communication. Kafka provides high throughput when compared with other messaging systems.

Kafka maintains the messages by writing to a file system, in contrast to other systems which maintain the messages in memory, and therefore the capacity is not limited by the RAM size.

Since messages are persisted on disk, it acts like a distributed commit log. The messages could be read by the consumer at any time and the consumer can also chose to read older messages by traversing back and forth through the message list.

Kafka uses Topics and each consumer can subscribe to one or more topics. Messages are written to the Topic by a producer and each topic maintains a commit log which stores the messages in the exact same sequence in which they were written. The index of the log is also known as offset.

Kafka works on a pull model. i.e., the consumers have to request for messages. Kafka keeps writing the messages as it arrives to a file and consumers have to maintain the index (or offset) of the messages being consumed. Thus Kafka transfers the overhead of tracking which consumer has read which message and helps in being lightweight.

Kafka supports partitioning and replication to provide high degree of scalability and redundancy. Each topic could be configured to have multiple partitions. The incoming messages are stored into one of the partitions. The Kafka cluster maintains the logs for a configurable period of time (retention period), after which the logs would be discarded.

Messages in a partition are also copies or replicated into other partitions, so that if one partition is down, the copy is available in other partitions. This architecture makes Kafka more suitable for distributed computing.

We can have cluster of multiple Kafka instances (known as brokers) and create topics whose partitions would be distributed across the broker instances. This would ensure high availability.

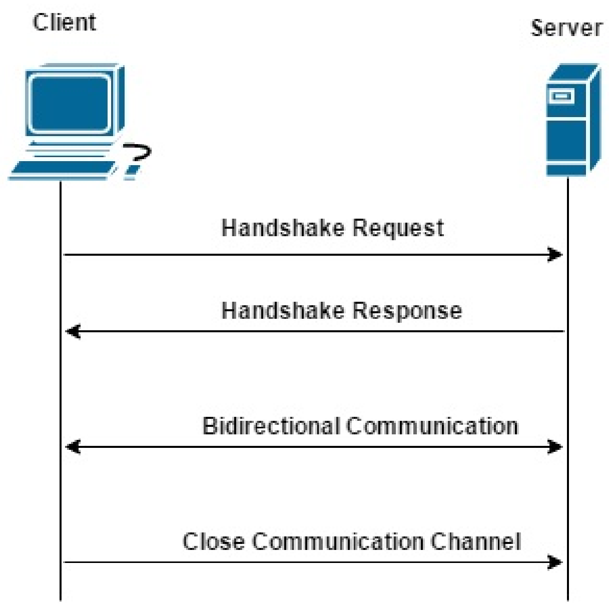
For example, let us say we have a cluster of 4 broker instances. If we create a topic T1, and configure 4 partitions for that topic, then the 4 partitions could be distributed across all the 4 broker instances.

Kafka uses zookeeper to maintain cluster state and also it keeps track of the index offset of each consumer. Kafka also provides a Streaming API to perform complex transformations on streaming data.

Kafka use cases:

Kafka is suitable for high throughput systems like data coming in from various sources like sensors, event streams etc. and consumers which would like to work through the messages offline or which would like to navigate through the messages.

WebSockets



WebSocket Communication Flow

In this section, we will take a look at WebSockets, a protocol, which is finding wider adoption in modern applications.

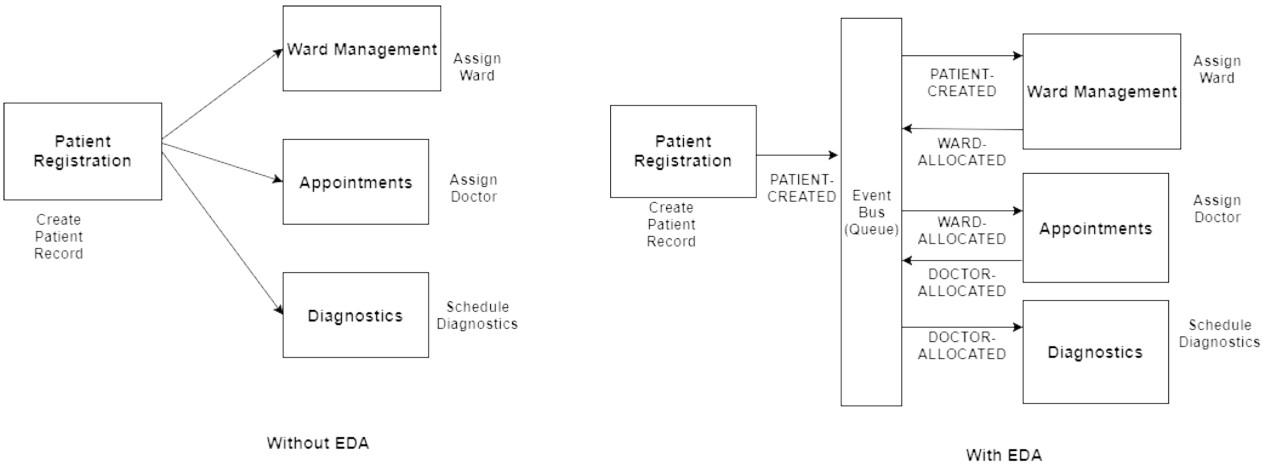
WebSocket is a bi-directional communication protocol based on TCP. Websocket allows the client (generally a browser) to open a TCP channel with the server and the server can continuously send data to the client. The client need not poll the server for data. This allows for real time message transfer between the server and the client over TCP port number 80. Since the standard port 80 is used, it can overcome the firewall rules which block other non-standard ports.

To establish a websocket connection, the client and server, exchange a handshake request which is similar to a HTTP request. (The request response headers consist of an Upgrade attribute, which could be interpreted by servers which serve HTTP requests).

Reference use cases:

Websockets are generally used in chat applications, applications which provide live stock updates etc.

Event Driven Architecture (EDA)



Event Driven Architecture
(EDA)

EDA is an architectural pattern which is based on system communication using events. The architecture facilitates communication by generating events and responding to events, thus providing a loosely coupled architecture.

For example, in a health care application, which is used by hospitals and diagnostic centers, let us take a look at the communication mechanism in the absence of EDA.

When a patient registers to be admitted to a hospital, there will be a series of actions like 'Create Patient Record', 'Allocate a ward room', ' Assign doctors and nurses' etc. Typically, there could be synchronous calls as shown below.

As we see in the above diagram, this system is tightly coupled and if one system is down, it adversely impacts the entire workflow. Also, addition of new modules becomes complicated.

Now, let us take a look at the same use case using Event Driven Architecture.

Creation of a patient record generates an event, say 'PATIENT-CREATED'.

There could be multiple consumers for this event. Two consumers consume this event and act accordingly as shown below:

These two consumers allocate a doctor; allocate a ward room respectively, upon consuming this event.

In this architecture, consumers and producers do not know each other. A message is placed on an event bus, with the message having supporting data for the event created.

For example, in the above example, the message could contain a patient_Id, the type of disease etc.

If the data is huge, we can place a correlation Id of the actual data and the consumer has to use another API call to retrieve the data. Once the event is consumed, the consumer can place another event (for example, DOCTOR-ALLOCATED), on the event bus, which would be subsequently consumed by other consumers.

Thus, events indicate 'State Changes' of a system and trigger other events. This pattern leads to a highly scalable loosely coupled architecture.

A mature Service Oriented Architecture uses EDA to achieve high degree of loose coupling.

Challenges of EDA:

Error handling:

If a consumer of an event is down, the message might have to be consumed again when the system is up again and back online.

We need a mechanism to play back events. For this to happen, we need to maintain event logs and a robust mechanism to play back events. Also, if an event is played again, the consumers which have successfully consumed the event should not process it again. For example, if ‘ALLOCATE-DOCTOR’ event failed and the first event is placed on the event bus again, 'ALLOCATE-ROOM' event should not process again. Thus, a lot of care has to be taken care to ensure integrity and consistency.

Devops

This section explains the various Devops concepts in detail.

Devops Principles

In this section, we will take a look at Devops, which is gaining wide popularity.

Devops is more of a movement or a culture, wherein the development team and operations work together without the corporate barriers. Managing the development team and operations separately in an organization hierarchy often gives rise to Developer vs Operations tussle. Though there is no single definition for Devops, there are a lot of processes and concepts which are usually associated with the Devops movement. Let us understand why do we need Devops and what are the challenges we face on a day to day basis?

Challenges faced:

1. Code shipped by development team, often would work in one environment, but not on others.
2. Development team would lack knowledge of deployment environments
3. Disparity of deployment environments could lead to a lot of issues. For instance, a database server might have different versions in different environments and this might cause issues during deployments, leading to delayed and painful releases.
4. Operations have little insight into development processes and internals; hence they only see their version of the story. As a result of separation of roles, we run into issues in deployment, blame games and quality issues and it would slow down the frequency of releases. These issues greatly impact business revenues in one way or the other.

How would Devops solve this problem?

The core solution lies in focusing on the end goal and our ultimate goal is that, we want to deliver the software to production with good quality. The barriers which come in way to achieve this goal should be torn down.

Devops brings together development and operations to work as one team. It brings about a change in mindset, allowing people to collaborate more, automate the processes and bring in parity in deployment environments. This leads to releases being stable and having good quality.

Processes associated with Devops:

There are few process aspects associated with the Devops movement to make it effective. They are listed down below:

1. Agile methodologies
2. Test driven development
3. Deployment processes
4. Branching and Release management Processes
5. Continuous Improvement
6. Automation using tools

1. Agile methodologies:

Use of Agile methodologies (there are lot of variations of Agile, which will not be covered in this book), which is a big deviation from the traditional water fall model of development, is a first definitive step. However, we will not go deep in this topic.

2. Test driven development:

Test driven development, is a very effective process in making sure that we develop quality code right from the beginning. Writing Unit test cases for code which we develop ensures that developers deliver quality code and also a sense of responsibility and ownership of quality. Quality is a collective responsibility of the entire team, and these processes will ensure that. There are a lot of unit test frameworks available today (like Junit, NUnit, Karma etc) and they are available for almost every programming language which we use today.

3. Deployment processes:

Deployment processes should be well defined and the team should adhere to that. Once the code is committed to a source control, code review, running of unit tests, security tests etc. should be performed on the code to ensure quality. Also, there should be gates which prevent bad quality code (which do not pass the adequate standards) from getting deployed. It should be the responsibility of the respective developer to fix bad code. Similarly, proper release notes should be published for every feature, to ensure deployment is smooth and hassle free.

Also, to ensure real Devops culture, each developer can also wear the hat of operations every now and then, for instance in every sprint (if Agile Scrum is followed) to understand the deployment processes. This way, every developer would take turns to deploy and would be cognizant of the environments, challenges etc. Understanding these would help them provide feedback and incorporate changes in the processes to make it better.

4. Branching and Release management Processes:

Branching strategy should be well defined and there are lot of distributed version control management systems (like Git), which allow geographically distributed development teams to collaborate and work. Code should be checked in very frequently and merged frequently (even many times a day). The Release management process should be well defined to allow frequent small releases (instead of one big release).

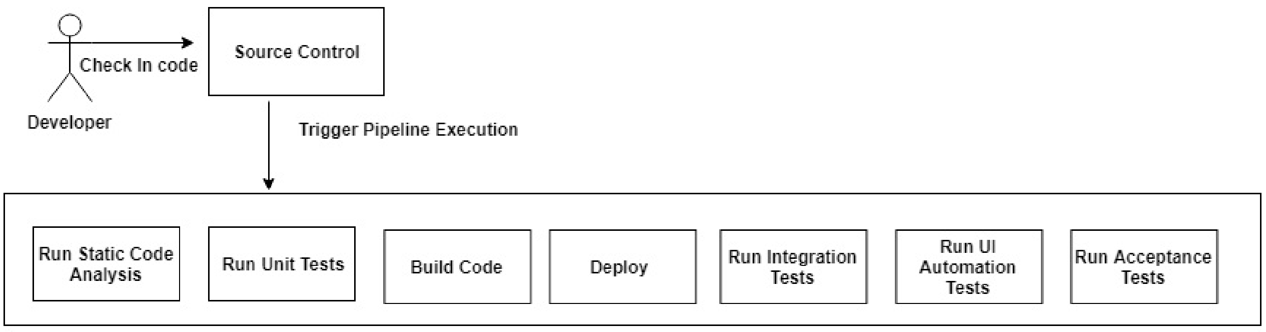
5. Continuous Improvement:

No process is fool proof and no process is perfect from day one. Also, a process which will work for one project, need not necessarily work well for the other. So, we need to have a continuous feedback loop, which should be immediately incorporated in the process life cycle and provide course correction.

6. Automation:

Automation of manual steps would lead to faster execution of the tasks and tasks being less error prone. It makes the end result more predictable. The commonly automated execution steps are:

- Build
- Code Reviews
- Unit Tests
- Integration Tests
- User Interface Tests
- Acceptance Tests
- Deployment



Continuous Integration

The above steps are collectively known as 'Continuous Integration' as illustrated in the above diagram.

There are various tools available for automating these steps and we can orchestrate these steps together into a pipeline, so that once the code is committed, all these steps are automatically run.

Deployment Automation:

Automation of deployment processes help in eliminating manual intervention and standardizes the process. Use of continuous integration and deployment processes help in having standard processes.

Infrastructure Provisioning:

'Infrastructure as a code' is a concept is which is gaining wide spread acceptance. Traditionally, Infrastructure setup is done manually and the steps are generally in the minds of few people who perform the deployment. Also, setting up new environments will be a Hercules task with a lot of legacy configurations, setup tasks and scripts. Often it consumes significant time and results in money drain.

Why is Infrastructure Provisioning important in Cloud?

With the advent of cloud, infrastructure is seen as a piece which needs to be setup and tore down quickly and on demand. Let us say, in a cloud environment, there are a huge number of services deployed, and we would need to scale out or scale down on the fly. We cannot manually configure each environment, since we will lose the ability to scale out quickly. Also, if there is a small change in environment configuration, or if we need to apply a small patch to software, manually applying that change would take a huge amount of time.

So we need a way to automate infrastructure provisioning. Infrastructure setup should be automated via programs or scripts and the entire datacenter could be setup this way. Any change to the environment, should be done through these scripts, which also should be version controlled. This way, the environment setup doesn’t depend on people, but rather would depend on automated processes.

This would help in cloud environments, where new environments (or virtual machines) should be provisioned on the fly. All we need to do is to run the scripts on new machines, and we get a new environment which will be an exact replica of existing environments. This will also eliminate human errors while setting up new environments.

Reference Implementations:

- Source Control revision management tools – Git, BitBucket
- Build Tools – Maven, Gradle

- Code Reviews Tools– SonarCube
- Unit Tests Tools – Junit, NUnit, Karma, Jasmine
- Integration Tests Tools – Citrus, Cucumber
- User Interface Tests Tools – Selenium
- Infrastructure deployment tools – Ansible, Chef, Puppet

Deployment Strategies

While deploying to cloud, the traditional application deployment techniques may not be sufficient. Traditional techniques like hot deployment or rolling deployment might work well, while deploying to a small cluster of machines. In traditional deployment techniques, since the application is not distributed, if an issue is discovered post deployment, it is relatively simple to perform a rollback.

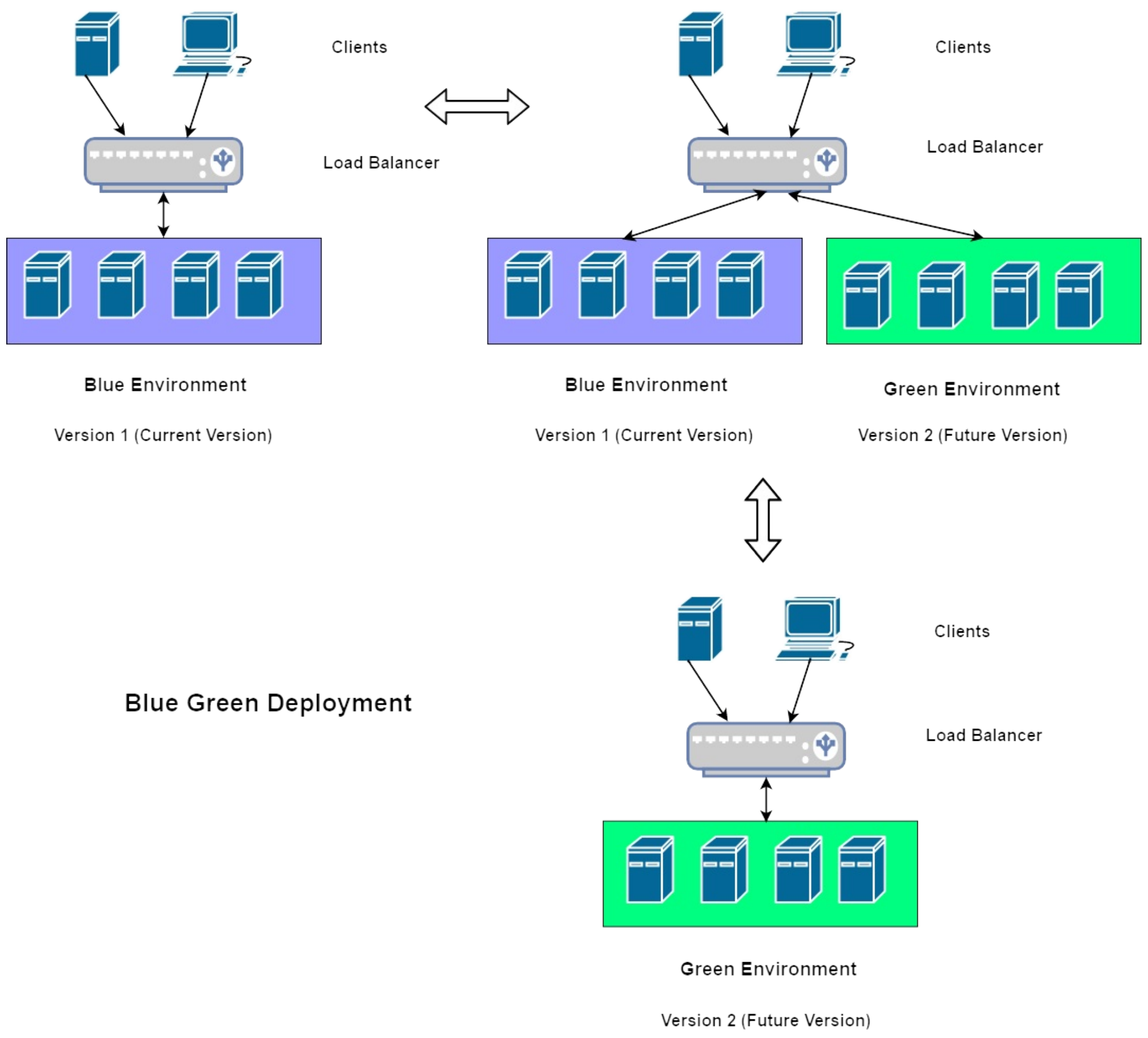
Imagine a cloud environment, which has hundreds of instances of a service/application. In such an environment, if we use the traditional approach of deploying to all the machines at once, just imagine, what would happen if an issue is detected post deployment? It would be nightmarish to perform rollback on each and every machine. Also, in a Microservices Architecture, there could be lot of services interacting with each other and rolling back may not be a trivial task. Obviously, we would need a modern cloud enabled deployment strategy.

Also, with continuous integration and deployment techniques, we are moving towards a shorter development to production cycle and hence increased frequency of releases. This, in turn, makes it more important to have an effective deployment technique which provides guard against any post deployment issues.

There are two popular Deployment Strategies:

1. Blue Green Deployment
2. Canary Deployment

1. Blue Green Deployment:



In this deployment technique, we will need to maintain two copies of the infrastructure during deployment, which could be named ‘Blue’ and ‘Green’.

Blue:

One part of the infrastructure (known as Blue) hosts the currently running version of the application.

Green:

We need to have a separate set of infrastructure (known as Green), which is identical to the other.

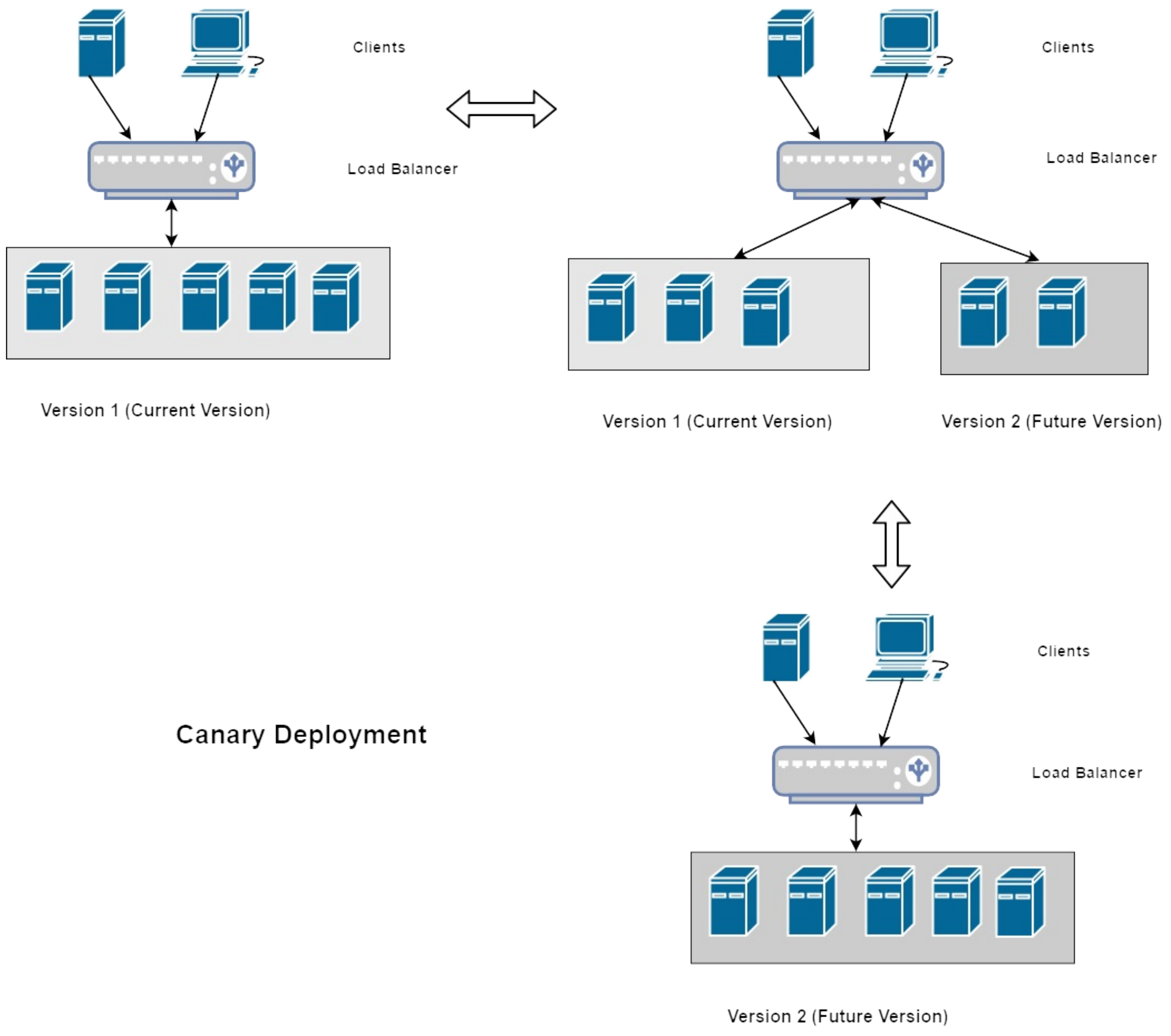
Deployment Process:

The new version of the application will be deployed on Green. At this point, we need to route the incoming requests to Green infrastructure. We need to retain this setup for a while and monitor and test the application. If we don’t find any issues, we can discard the Blue environment. If something is wrong, then we need to switch the requests back to Blue. With cloud enabled infrastructure and automated deployment, we can quickly setup/tear down new environments. Post successful deployment, the Blue environment could also be used for Staging/UAT testing.

Challenges:

1. While switching from Blue to Green or vice versa (during rollback), if there is a transaction which is in progress, we need to gracefully handle those transactions and allow them to complete.
2. Database changes: If there are significant changes to database in the new release, it may not be easy to switch requests between Blue and Green. One of the solutions to this is to use DB refactoring. DB Refactoring is a process of making changes to a database so that it supports both older and newer versions of the software.
3. Requires two copies of infrastructure during deployment time and could be an overhead.

2. Canary Deployment:


Clients
Load Balancer
Version 1 (Current Version)
Clients
Load Balancer
Version 1 (Current Version)
Version 2 (Future Version)
Canary Deployment
Clients
Load Balancer
Version 2 (Future Version)

Canary got its name from a bird which is used in coal mines to check for the presence of toxic gases.

In this technique, the new version of software is deployed to a small subset of servers (infrastructure). A small subset of user requests is routed to this subset of servers. Typically, users could be selected by geographical location, or based on division/company to which the user belongs to. The application is then tested for sanity. If everything is fine, then slowly, the new version of software is deployed to larger set of servers is incrementally (and routing larger set of user requests to it) until all the servers are covered.

Challenges:

1. As in Blue/Green, managing Database changes is a challenge, especially, if the database changes are disruptive (involving major schema changes).

Security

This section explains the various Security concepts in detail.

Encryption and Hashing

In this section, we will take a look at two concepts which are widely used in Computer security.

Encryption:

Encryption is a cryptographic operation which transforms a text to a seemingly jumbled or a random text and decryption is an operation of retrieving the original text from the encrypted text.

There are two popular Encryption techniques:

1. Symmetric key Encryption In this technique, the key used for encryption and decryption are the same.
2. Asymmetric key Encryption (Public/Private Keys) In this technique, a different key is used for encryption and decryption. Generally, the encryption key is made public (hence the name) and the decryption key (private key) is held privately by the encrypter. Note that, specific algorithms are available to generate a set of Public and Private Keys.

Importance of Public/Private Keys:

Public/Private Keys are used in a variety of applications, the most important being to secure message transmission. In general, this concept (as defined below) is used in implementing transport layer security (e.g., HTTPS), PGP etc.

Use of Encryption in secure message transmission:

In message transmission, public/private key encryption is generally used. The receiver of the message would generate a set of public/private keys and would make the public key known to everyone (so anybody can encrypt and send messages) and the decryption key is held securely with itself. Thus, when a message has to be relayed to the receiver, the sender would encrypt the message using public key. The receiver would decrypt the message using private key.

Below is a list of algorithms which provide Public/Private key cryptography:

1. RSA
2. Digital Signature Algorithm

Hashing:

Hashing is a mathematical operation which transforms a text or a set of characters into a different text or set of characters. The mathematical function which performs this operations is known as a Hash function. The result of a hash operation is known as a hash result or just ‘Hash’ value.

An ideal hash operation is irreversible. i.e., we cannot derive the original text by operating on the hash result.

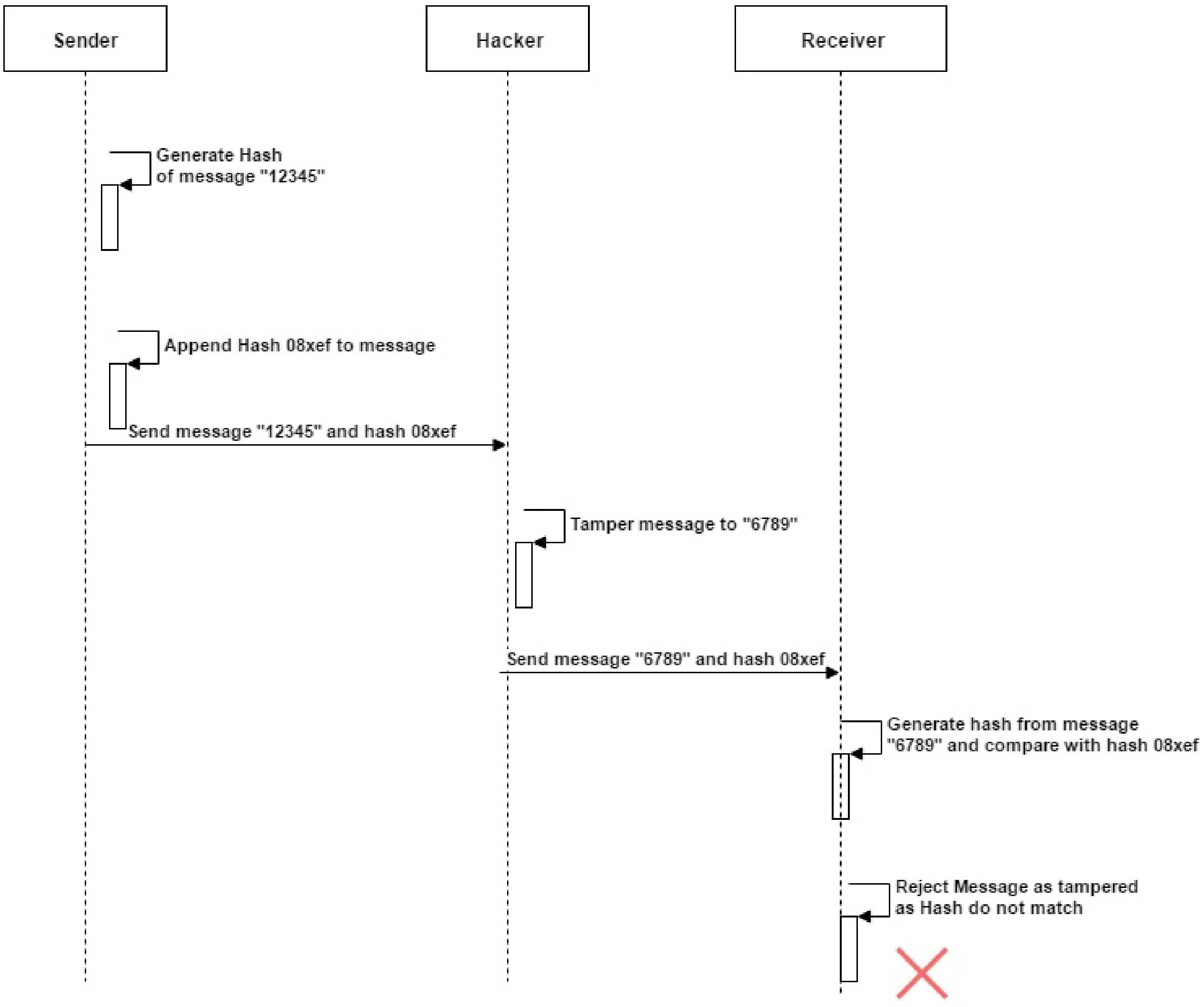
Salt:

Salt is a random data that is used as an additional input to the hash function. It would help in generating hashes which are difficult to guess or reverse engineer the hash to retrieve the original text.

Importance of hash in security:

In the software security world, hash plays an important role as illustrated below.

1. Securing message communication against tampering the message in between:


Sender
Hacker
Receiver
Generate Hash of message "12345"
Append Hash 08xef to message
Send message "12345" and hash 08xef
Tamper message to "6789"
Send message "6789" and hash 08xef
Generate hash from message "6789" and compare with hash 08xef
Reject Message as tampered as Hash do not match

Hash usage in Secure Communication

Let us say system A is sending a message to system B by some means (like HTTP communication). A hacker can intercept the transmission in between and tamper the message. In order to prevent this scenario, the hash could be used in the following way.

- The sender would generate a hash of the message using a secure key which is only known to the sender and the receiver.
- The sender would send the message as well as the hash to the receiver.
- The receiver would retrieve the message and the hash. The receiver would generate its own hash from the message and the key (which was agreed upon between sender and receiver).
- The receiver would compare the hash which it generated in step c with the hash sent by the sender. If both of them match, then the receiver can conclude that the message has not been tampered with. If the hashes do not match, then the message has been tampered with.

Since, the hacker will not know about the secret key required while generating the hash, the message cannot be tampered with, or else the receiver would come to know about it.

2. To store passwords for authentication:

Generally, application user passwords were being stored in an encrypted format. However, with the advance in technology, it is becoming far easier for powerful systems to decrypt the encrypted passwords. Thus, if a hacker gets hold of the database having encrypted passwords, then decrypting them would lead to compromising secure accounts. To overcome this, hashing is generally used.

- The password is first hashed and the resulting hash is then encrypted and stored in database.
- Even if a hacker gets hold of the encrypted value and decrypts it, the hacker cannot retrieve the original text from the hash.
- During login, when a user provides username/password for authentication, the system would first decrypt the value from database and retrieve the hash. The system would then compare the hash of the password provided by the user and the hash retrieved from the database. If they match, the authentication is considered successful. Note that in this setup, there is no way to retrieve the original password from the system.

The popular hash functions used in security include:

1. MD2
2. MD4
3. MD5
4. Secure Hash Algorithm (SHA-1, SHA-2)

Generally, Secure Hash Algorithms (SHA-1/SHA-2) along with a random salt are used to hash passwords and store it in database. Note that the salt should be randomly generated and should be different for each user and stored along with the hash in the database. Creating a random salt per user would add additional security and guard against rainbow table attacks.

Whitelisting and Blacklisting

In software security, we often come across two ways of allowing or blocking resources.

1. Whitelisting
2. Blacklisting

In this context, a resource might refer to an application, a URL, a domain etc.

Whitelisting:

Whitelisting is a process wherein a set of resources are granted explicit access. For e.g., If we whitelist domains google.com and yahoo.com, then these two domains are always allowed by default.

Blacklisting:

Blacklisting is a process wherein a set of resources are blocked explicitly. For e.g., if we blacklist domains hacker.com and malicious.com, then these two domains are always blocked by default.

Also, a combination of whitelisting and blacklisting could be applied.

Whitelisting Vs Blacklisting:

Both of these approaches have pros and cons.

- Blacklisting will help to defend against known malicious resources, but cannot defend against new malicious resources. For e.g., what if an application/domain which is extremely harmful makes its presence on a certain day and our blacklist will not have that information until it is known and explicitly configured. By that time, the damage would have already been done. Whitelisting on the other hand defends well against these types of new unknown resources.
- Whitelisting, if not done properly, might interfere with the day to day working of users. For e.g., if some corner usecase of a known application needs access to a different domain than which is configured, then it might cause disruptions in regular work. However, a well-planned whitelist is always considered as secure.

Reference Use cases:

One use case of applying whitelisting and blacklisting is applying it for URLS in network layer tools. Another use case is applying it in an application for acceptable characters for input fields for protecting against attacks like XSRF etc. For instance, certain fields like name field or address field on web pages could be restricted to accept only certain characters (whitelist) and prevent special characters (blacklist).

Ingress and Egress Filtering

In network security, two types of filtering namely, Ingress and Egress are used to secure the network.

Ingress Filtering:

Ingress filtering is a process of monitoring and restricting the flow of information (or packets in networking terms) which enters a network.

Ingress filtering is used to prevent the network from being attacked by malicious sites and also to prevent denial-of-service attacks.

Ingress filtering involves examining the packets to verify the source IP address and check if they are genuine.

Egress filtering:

Egress filtering is a process of monitoring and restricting the flow of information (or packets in networking terms) going outside of a network.

Egress filtering is used to prevent malicious or unauthorized content from leaving the network. It also ensures that only supported protocols are used in the outgoing traffic.

Egress filtering could be applied by using a proxy server to route the outgoing traffic. The traffic from all the servers/systems in the network should go through the proxy. Egress rules could be applied on the proxy server.

Web Security

From a Security perspective for web applications, there are a lot of aspects to consider other than using a HTTPS communication. Using HTTPS provides transport layer security, but from an application security perspective, there is more to ponder over. There a lot of tools which perform tests (often known as Penetration tests) on the application, scan the code and provide possible vulnerabilities.

Open Web Application Security Project (in short referred to as OWASP) is a not-for-profit charitable organization focused on improving the security of software. They provide a list of common vulnerabilities and possible solutions for each.

Following is a list of some of the common attacks which needs to be generally considered.

1. Injection Attacks

Injection attacks involve modifying the input to an application in such a manner that the input itself could be used to perform malicious activity on the system. SQL Injection attack for instance includes modifying the input to embed SQL queries into the input which either gather sensitive information or perform harmful operations like deleting data, dropping tables/databases etc. JSON injection attacks also involve modifying the JSON to perform malicious queries on JSON based databases.

Illustration:

A simple SQL injection attack could be tried by modifying the input as below:

Solution:

Solution to injection attacks involve sanitizing the input against specific patterns and also avoid using the input to concatenate and create queries.

2. XSS Attacks

Also known as Cross Site Scripting attacks, in this type of attack, the attacker exploits the vulnerability of an application which just renders all the inputs on its web pages without sanitizing it. For eg, an attacker would provide a malicious script as his/her profile input to a website (say social networking site) and lets say the website redisplays this input on its website without sanitizing it. So, when another user visits the attacker’s profile, the script is rendered by the website and the script is run by the browser on the user’s machine. This script could steal sensitive user’s information and transfer it to the attacker’s site.

Solution:

Solution to XSS attacks involve sanitizing the input by validating against white listed set of characters and validating against script inputs.

3. XSRF Attacks

Also known as Cross Site Request Forgery attacks, in this type of attack, the attacker tricks the users to perform malicious operations on a site without the user’s knowledge. The malicious operation may or may not benefit the attacker, but nevertheless could harm or damage the user’s reputation.

For eg, Let’s say a user is logged in to a shopping cart application. An attacker would show some deals related to the shopping cart on his own website and trick the user to click on his website link. When the user clicks on the attackers website, the attackers website would run the same javascript code (form submission) which is actually present in the shopping cart application, but with malicious values. Since the user is already logged in, probably in another tab of the browser, the session would be shared and the attackers form submission could succeed. The attacker could possibly place a new order or change the delivery address and update it to his address etc.

Solution:

Solution to XSRF attack involve embedding a time limited server side random token or some form of signature in the web pages which would be submitted along with every request and validated against forged submissions.

4. Denial of Service Attacks

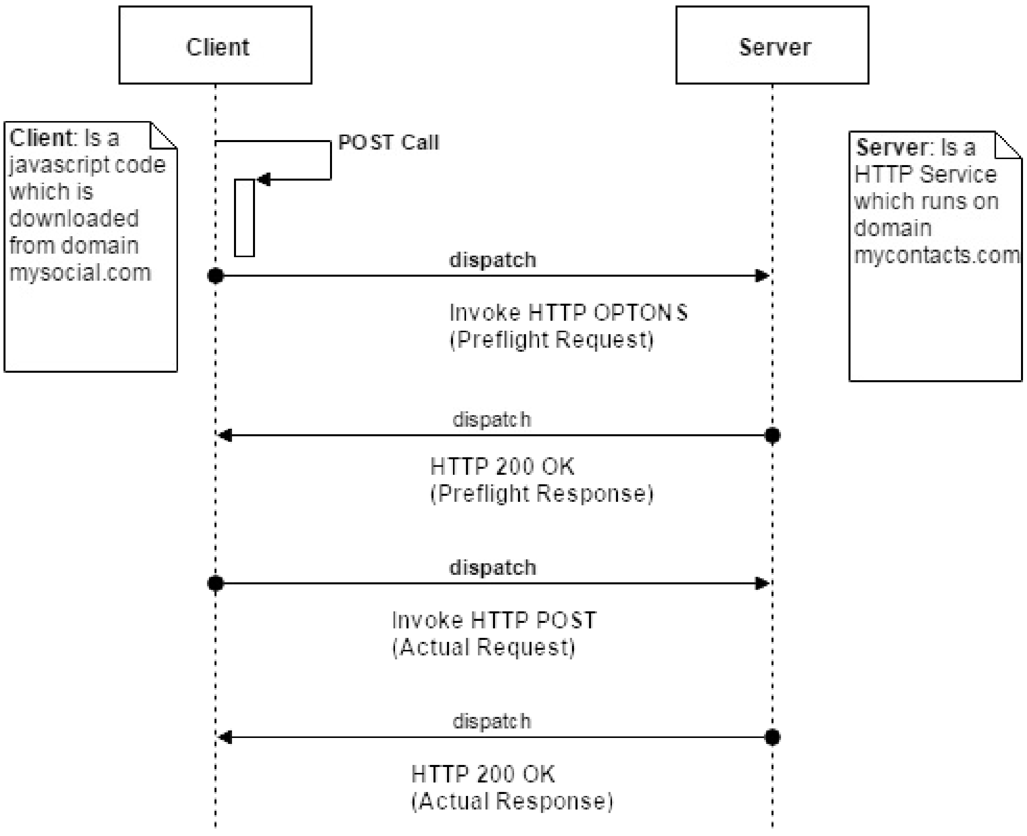
Denial of Service attacks (in short DOS attack), involves pumping a server with a huge of number of requests (which are automated), till the server gets overloaded and goes down. Another variant of this known as Distributed Denial of Service attack (DDOS), involves pumping of requests originating from different distributed attack servers (with unique IP addresses), making it hard for the vulnerable server to identify and block the attack requests.

In a cloud based environment, denial of service attacks might lead to scaling of infrastructure (if it is configured for auto scaling) and could incur huge financial losses for organizations.

Solution:

There are commercial softwares available which identify and neutralize DOS attacks.

CORS


Client
Server
Client: Is a javascript code which is downloaded from domain mysocial.com
POST Call
dispatch
Invoke HTTP OPTONS (Preflight Request)
Server: Is a HTTP Service which runs on domain mycontacts.com
dispatch
HTTP 200 OK (Preflight Response)
dispatch
Invoke HTTP POST (Actual Request)
dispatch
HTTP 200 OK (Actual Response)

CORS Requests

In this section, we will take a look at the concept of CORS, which has a significant role to play in modern web applications.

CORS stands for Cross Origin Resource Sharing.

Before understanding CORS, let us take a look at Cross Origin Requests. In HTTP, a Cross Origin request happens when a resource, which belongs to a particular domain, makes a request to a resource which resides in another domain.

For e.g., let us say we are visiting a site mysocial.com. A javascript file which is rendered from the domain mysocial.com makes a HTTP request to a domain mycontacts.com. The original domain, from which the javascript resource was rendered from, is not the same as the domain to which it makes a HTTP request. These kinds of requests could be classified as Cross Origin requests.

Historically, browser implementations were preventing the Cross Origin requests for security reasons. But with the recent advances in UI technology (and the way applications are built), Cross Origin requests look inevitable, and a new standard was required to allow Cross Origin requests.

W3C came up with a standard way of allowing Cross Origin requests, known as ‘Cross Origin Resource Sharing’.

Note that including image urls from a different domain in an html page doesn’t come under Cross Origin requests. Generally, Cross domain AJAX requests, HTTP request methods from libraries like Angular etc. come under Cross Origin requests.

How does CORS work?

As part of CORS specification, a few simple Cross origin requests are allowed by default. For e.g., GET, HEAD, POST requests (there are some MIME type constraints as well for these requests) are allowed based on the MIME types. For other type of requests, whenever cross origin request is made by a script, the browser has to send a pre-flight request to the corresponding server (i.e, to mycontacts.com server in above example).

A Preflight request is a HTTP OPTIONS request sent by the browser to the server, along with details like origin server (which is the original server url to which the resource belongs to), the type of HTTP request being attempted (i.e. GET/POST etc.), Content type etc.

While using libraries like Angular JS and making Cross Origin requests, if we inspect the Network calls made, we can see a lot of OPTIONS requests made to the server. These OPTIONS requests are the Preflight requests sent by the browser.

The server should respond with a 200 status response for this Preflight request (if the origin domain is allowed and secure) with additional headers in the response like Access-Control-*. (Access-Control-Allow-Origin, Access-Control-Allow-Methods etc.).

If the server responds with a 200 OK response, then the browser sends the real request to the server. The above diagram illustrates the flow explained here.

Note that to allow CORS, browsers need to fully support CORS standard and the server needs to support the CORS request.

Generally, on the server side, a HTTP request filter is required to allow the server to respond to the preflight OPTIONS request. As a good practice, the server can also whitelist the domains from where the Cross domain requests should be allowed instead of allowing a blanket entry for requests from all domains.

Application Architecture and CORS:

Generally, in a Microservices Architecture, there could be a lot of services exposed on different domains. The UI clients based on javascript libraries (like Angular etc.), could invoke these services and they invariably end up with making Cross Origin Requests. Implementing a secure CORS on the server side is very essential in such a scenario.

OAUTH

OAuth is security protocol, which is used to grant authorization access resources.

From the wiki, "OAuth is an open standard for authorization, commonly used as a way for Internet users to authorize websites or applications to access their information on other websites but without giving them the passwords".

The current version of OAuth at the time of writing this book is OAuth2.0

Roles in OAuth2.0:

There are four roles in OAuth2.0:

1. Authorization Server
2. Resource Server
3. Client
4. Resource Owner

Let us take a look at each of these.

1. Authorization Server:

Authorization Server is the server which performs the following actions:

- Authenticates the client
- Issues a time limited token
- Validates the token (including checking the token for expiry)

2. Resource Server:

Resource server is the server which hosts the Resource in question, and provides secure access to the resource. (By validating the token provided in the access request)

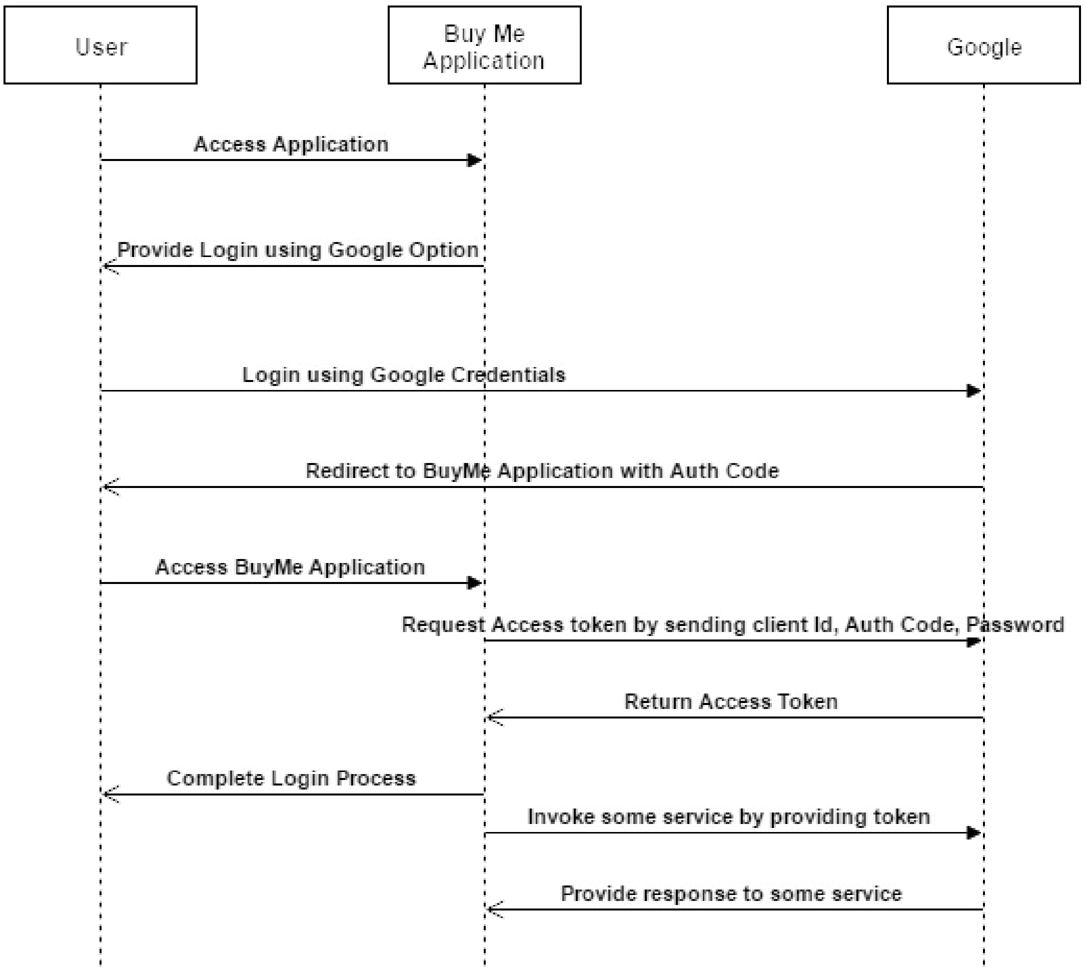
3. Resource Owner:

Resource Owner is the user who authorizes the access to its account.

4. Client:

Client is the user/application which requests access to the resource.

Example Use case:


User
Buy Me Application
Google
Access Application
Provide Login using Google Option
Login using Google Credentials
Redirect to BuyMe Application with Auth Code
Access BuyMe Application
Request Access token by sending client Id, Auth Code, Password
Return Access Token
Complete Login Process
Invoke some service by providing token
Provide response to some service

OAUTH 2.0 Workflow

1. Let us say a user accesses an ecommerce application, 'BuyMe'. The application 'BuyMe', presents a login option to sign in using Google/Facebook Account.
2. The user selects 'Login using Google Account'.
3. The user is redirected to Google Login page.
4. When the user logs into Google Account, the user is asked whether he/she would like to allow 'BuyMe' application to access few details from his/her Google Account.
5. Upon agreeing to grant access, the user would be redirected to the BuyMe application.

There are few things going on behind the scenes in the above use case:

1. Provisioning of the Client application with the Authenticator:

The BuyMe application would have registered itself initially with the Google Application for the purpose of Authentication. During registration, the Client Application would have provided the Authenticator (Google App) with a redirect URI and, the Authenticator would have provided the BuyMe application with a Client Id and a secret. This is a onetime process also known as provisioning.

2. Authentication Code:

The Authenticator (Google App) would append an Authentication Code to the URI while redirecting to the client application. The client application would request for a token from the Authenticator (Google App), by sending the client ID, password and the authentication code. The authenticator would validate these details, and if valid, would send back a token. The client application would send the token in all the further requests while accessing data from the Google application.

Grant Types:

OAUTH 2.0 specifies four types of Grants:

1. Authorization Code
2. Implicit
3. Resource Owner Password Credentials
4. Client Credentials

1. Authorization Code

The above use case illustrates 'Authorization Code' type Grant.

2. Implicit

In the Implicit type Grant, the Authorization code is not returned. Instead, the token is immediately returned to the client application.

3. Resource Owner Password Credentials

In 'Resource Owner Password Credentials' grant, the client application maintains the username/password of the Authenticator application and the client application would use these credentials to send them over to the Authenticator.

4. Client Credentials

In the 'Client Credentials', the client is requesting access to protected resources under its control (i.e. there is no third party).

Why is OAuth important in the cloud?

In cloud environment, if Microservices Architecture is embraced, then we need a way to secure all the individual services. OAuth is a popular choice for securing these services, with each service acting as a 'Resource Server'. The Auth Server could be a home grown server, which could be built to authenticate the user and provide a time limited token.

Also, with a lot of modern B2C applications allowing multiple login options (like login via Facebook, Google etc.), OAuth protocol, invariably finds an important role to play.

SAML

In this section, we will take a look at SAML, which is a popular security protocol, used very widely.

Security Assertion Markup Language (SAML) is an XML-based, open-standard data format for exchanging authentication and authorization data between parties, in particular, between an identity provider and a service provider. SAML is a product of the OASIS Security Services Technical Committee.

SAML is generally used in implementing Single Sign On. Single Sign On is generally used in intranet applications in a corporate network, wherein, a user can login to a single application and navigate to other applications seamlessly without having to login to each application

SAML Concepts:

1. Identity Provider (Idp)
2. Service Provider (SP)
3. Principal
4. Assertions

1. Identity Provider (Idp):

An Idp, maintains a list of valid users of the system and provides a service to authenticate an user. This process of authentication is generally termed as ‘Assertion’. As part of the assertion, the Idp also provides information it has about the user (or the principal). An Idp also provides a service which validates the authenticity and the validity of a token.

2. Service Provider (SP):

A SP, provides the actual service (or is the application) which would be used by end users.

3. Principal:

Principal is the actual end user or a client system (who provides credential for authentication) and consumes the services provided by the Service Provider.

4. Assertions

Assertions are packets of security information exchanged between Identity Provider and Security Providers.

SAML is generally built upon XML and SOAP specifications and exchange of information happens through digitally signed XML documents, thereby making it secure.

SAML Flows:

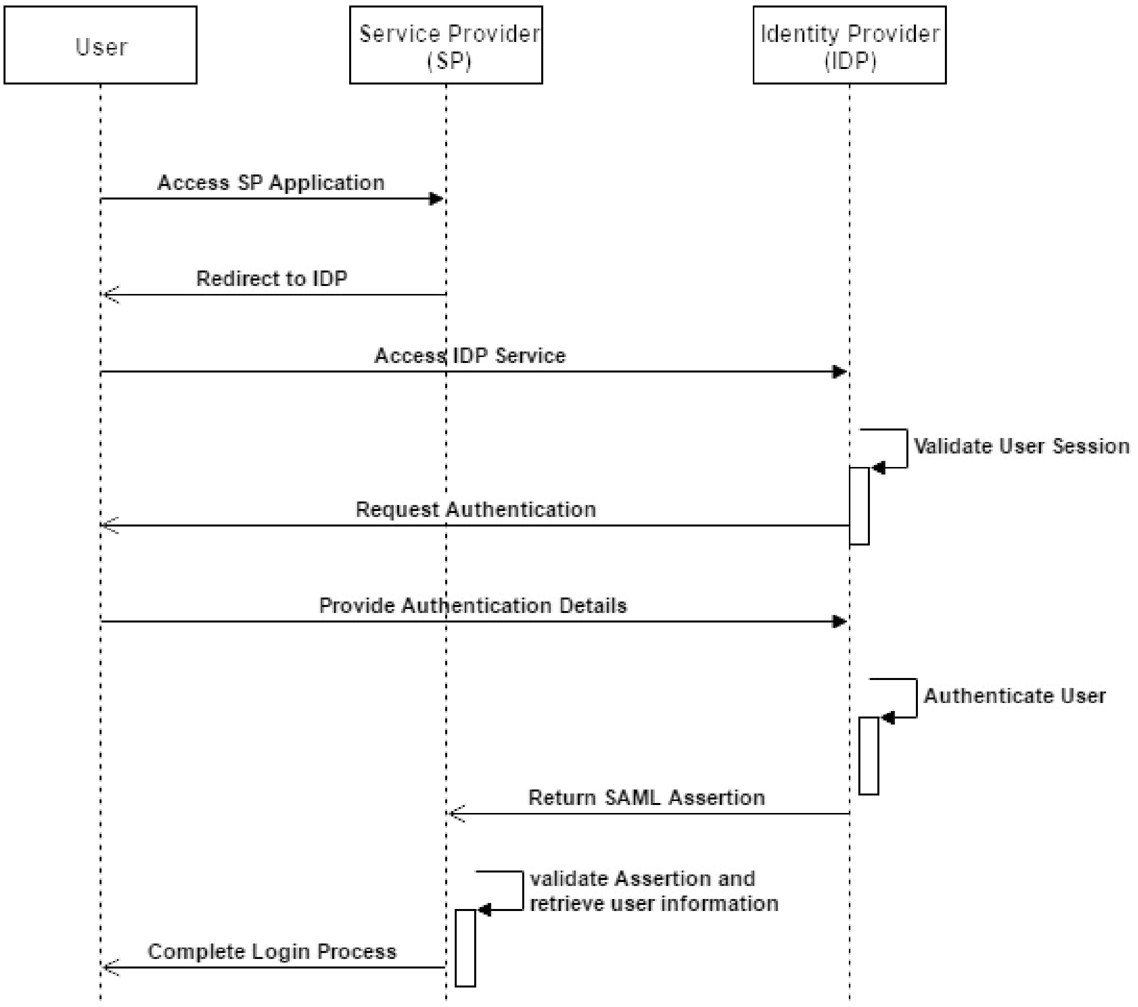
There are two common flows associated with SAML.

1. SP initiated SSO
2. Idp initiated SSO

A common step is to have the Identity Provider and Service Providers establish themselves and this process is known as Provisioning. This is a one-time activity.

Login Scenarios:

The below diagram illustrates SP Initiated flow for simplicity.



SAML Login Workflow

1. SP initiated SSO:

1. A user clicks on an application link which is rendered by a Service Provider.
2. The SP application redirects the user to an Identity Provider, asking for authentication.
3. The Identity Provider, checks if a valid session for the user has already been created and is active. a. If there is no such session, the Idp asks the user to authenticate. After successful authentication, the Idp creates a session and returns a SAML assertion with the user details. b. If a valid session is already created, then the Idp returns a SAML assertion with the user details.
4. The SP receives the authentication assertion from the Idp and validates the response.
5. The SP allows the user to access its services.

2. Idp initiated SSO:

1. A user clicks on a portal page which is rendered by the Identity Provider.
2. The Identity Provider, checks if a valid session for the user has already been created

and is active. a. If there is no such session, the Idp asks the user to authenticate. After successful authentication, the Idp creates a session and returns a SAML assertion with the user details. b. If a valid session is already created, then the Idp returns a SAML assertion with the user details.
3. The authenticated user then clicks on any of the links provided by the SP.
4. The SP receives the authentication assertion from the Idp and validates the response.
5. The SP allows the user to access its services.

So, in general, Idp initiated SSO is used if the user lands on a well-defined login or portal page and then moves on to access any of the links provided by the SP. SP initiated SSO is used, if the user clicks on any of the deep links provided by the SP (without going through the initial portal login page).

Logout Scenarios:

SAML also supports logout scenarios from multiple applications as explained below. The below examples correspond to a web SSO application flow.

1. SP initiated Logout:

1. A user, who is logged into a SP, clicks on a logout button.
2. The SP terminates the session and initiates a logout request to the Idp.
3. The Idp finds out how many SP sessions are maintained by it.
4. If there is only one SP (the one which initiated logut), then it sends back a logout response.
5. If there are other SPs whose session is valid, then the Idp sends out a logout request to each of the SPs.
6. The SPs respond back with a logout response.
7. The Idp terminates the session maintained by it for this user and sends a logout response to the SP who initiated the logout request.

2. Idp initiated Logout:

The Idp initiated logout scenario is similar to the above, except that it starts from step 3. It would be initiated when the user clicks on a global logout link provided in the portal or if the Idp session expires.

Reference Use cases:

SAML is used in implementing Single Sign on, for related applications. A common use case is Single Sign on for Employee related applications in a corporate, like Leave management system, Employee Profile application etc.

HTTPS Security

HTTPS (Secure HTTP), as the name suggests is secure way of communicating over HTTP protocol. Under the hood, it uses a cryptographic protocol known as SSL (Secure Socket Layer) or TLS (Transport Layer Security), which provides encryption of data.

Note that SSL protocol is deprecated and is replaced by TLS protocol. There are various versions of TLS protocol which could be used for example, TLS 1.1, TLS 1.2 etc.

We will be using the terms SSL and TLS interchangeably in this section to provide general overview of how they work.

A SSL communication happens between two systems, referred to as a Client and a Server. There are two categories of system interaction:

1. Client is a Browser, and the Server is an application running on a web server.
2. Client and the Server are both applications running on a web server.

Further, there are two types of SSL Communication:

1. One way SSL - Only the server has a valid certificate and this certificate is used in the communication.
2. Two way SSL (or Mutually Authenticated SSL known as MASSL) – Both the server and the client have valid certificates and these certificates are used in the communication. In MASSL, the server would be configured to mandatorily expect a certificate from a client.

Certificates and Authority:

We would look what is a Certificate, who issues a Certificate etc. in the below sections.

Certificate:

A Certificate provides a proof of identity to a system (the system can act either as a client or a server in a SSL communication). To give an analogy, a certificate could be an Identity proof for an individual like a Passport for instance.

Certificate Authority:

Certificates are issued (actually signed) by well-known and trusted organizations like Digicert, Verisign etc., which are known as ‘Certificate Authority’ (CA). Continuing with the analogy, certificate authority are like governments who are authorized entities to issue a passport.

Certificate Trust Store:

A trust store is a logical repository of certificate Authorities, which a system (a browser or a server) would have. Browsers generally will have a list of trusted CA's pre-installed in their trust store, and we can also add additional CA's into the browser's trusted CA store.

Similarly, we can initialize and configure trust stores in web servers (for example, Nginx, Apache etc.), load balancers or any system which takes part in a SSL communication.

Certificate Chains:

There are two types of CAs:

1. Root CA
2. Intermediate CA

The intermediate CA would have a certificate which is issued and signed by another intermediate CA or a Root CA. Thus we can have a chain of certificates. For example, example.com could have a certificate which is signed by intermediate.com and intermediate.com can have a certificate which is signed by Verisign (which is a Trusted Root CA).

A browser (or any system) would look through the chain of certificates until it finds a certificate which is signed by a trusted CA.

How do we generate a Certificate?

A certificate is associated with a public key and a private key, to facilitate asymmetric encryption. We need to generate a private key and a public key using various tools like (OpenSSL for example) and generate a CSR (also known as Certificate Signing Request) using tools like OpenSSL.

A CSR is a file which contains the public key and the various details like the Organization who is requesting this, the domain name in which the certificate would be installed etc.

This CSR needs to be sent to the CA (the private key should not be sent) and the CA would generate a certificate and digitally sign it.

The certificate thus obtained along with the private key, could then be installed on servers.

Self-Signed Certificates:

Self-signed certificates are certificates which are not issued by trusted CA's but issued and signed by individuals or organizations on their own.

Why do we need Self signed Certificates?

Let us say an organization ‘XYZ Corporation’ has multiple applications which need to interact amongst themselves using SSL. Getting a certificate using trusted CAs would involve cost and in such cases, the organization can issue certificates which are signed by an Authority within the organization. These certificates could be used for any communication within the Organization, but cannot be used for communication with parties outside the organization, as they would not trust this Certificate Authority. Similarly, these certificates cannot be used for browser to server communication, as browser would not have this CA as a trusted CA. Generally, if you try to browse a website which has self-signed certificate, the browser would show a warning indicating that the website is not secure.

SSL Handshake:

Handshake is a mechanism by which the client and the server exchange information, check each other’s compatibility and agree upon certain things. Let us see the handshake mechanism in more detail.

1. Exchange Hello:

The client sends ‘client hello’ message, in which it sends details like, the TLS protocol it supports, the various cipher suites and a list of algorithms it supports. The server responds with a ‘server hello’, in which, it sends the minimum TLS protocol version required, cipher suites and the list of algorithm it prefers to use.

2. Exchange Certificate:

The server would send its certificate information, which would contain, the public key, certificate Issuing Authority (CA), its validity period. The client would check, if the server has a certificate which is issued by an authority which it trusts and is valid. Every client would have a list of trusted CA’s and if the server certificate’s CA is a CA which is trusted by the client, then handshake will proceed.

If the client also has a certificate (explained later in MASSL), then the client would present its certificate to the server for verification.

3. Exchange Key:

Once, the protocol and algorithms are agreed upon in step 1, and the client has trusted the server (in step 2), the client and server need to agree upon a secret key which would be used for encrypting the future messages. The client would generate a secret key (symmetric key), and it would encrypt the secret key with the server’s public key (obtained in step 2) and send it to the server. The server would decrypt the secret key using its private key and establish a master secret key to be used for all further communications. A SSL session would be established between the client and server and the master key would be valid only for the duration of the session.

Thus, both the server and the client exchange information by encrypting the data using the master key. Thus, a hacker who would try to eavesdrop into the channel and find out what information is being exchanged, would not be able to do so, since the data is encrypted. Also, the key used for encryption is specific to the session established, thus providing enhanced security.

MASSL:

In MASSL, the client is also expected to have a valid Certificate, just like the server does, and in step 2 above, the client is also expected to send its certificate to the server. The server would validate the certificate to establish the identity of the client.

MASSL is used in situations, wherein, we would want only known clients to communicate with the server. MASSL would provide additional protection, wherein untrusted clients cannot communicate with the server. MASSL is generally used to secure communication between two systems and is not generally used if the client is a browser or a browser based client.

Putting it all together

There is no architecture which fits every problem and no architecture is perfect. Architectural decisions always involve tradeoffs and one has to decide and choose one over the other. For example, the tradeoff could be between performance and security or performance and usability etc. An architect has to carefully weigh multiple options and choose the right solution which resolves the problem at hand.

The next section describes a sample usecase and explains how an architect would go about building a solution, by using the various concepts which are covered in this book. The concepts which are covered in this book are represented in bold. I will not attempt to provide a solution, but emphasize on the thought process which is used to derive a solution.

Reference UseCase

Let us say an Architect Kris, has been asked to engineer a solution for a customer facing application which accepts feedback for various products from the customer. While there will be other folks to capture the functional requirements (which, Kris will also be across), Kris will be mainly focusing on the **Non functional requirements** of the application.

Few of the key non functional requirements, which catches his eye are:

1. **Application availability** - 99.99.
2. Response time - 10 ms
3. Peak **Transactions per second (TPS)** - 400

Apart from the functional and non functional requirements, Kris also gathers information about the business needs of the application and comes to know that the application has to evolve constantly to discover and implement newer ways of gathering feedback, to stay competitive. For example, the application might have to mine information from social media, or gather indirect feedback by using analytics which could interpret customer actions.

Methodology:

Kris realizes that the application would need frequent release cycles, as short as every week to introduce new features. So, he decides that following Agile practices and following **Devops methodology** could prove to be very effective.

Target Platform:

Kris, needs to choose a target platform which can provide the availability criteria specified in the non functional requirements and design a solution which will guarantee this availability. He choses the right cloud platform by examining the various **factors which are involved in choosing a cloud platform**. Kris analyzes the various **Infrastructure As a Service** offerings and **Platform As a Service offerings** to choose the right solution which fits the needs. Based on the application availability specified, Kris calculates **RTO and RPO** and comes up with a **disaster recovery** plan to ensure that the availability requirement is met.

Application Architecture:

Kris also realizes the need for adopting a modern architecture (like a **Microservices Architecture**), in order to build loosely coupled services quickly which can evolve over time. In order to meet the response time, he finds that building a loosely coupled scalable architecture is necessary by using **asynchronous communication** channels wherever applicable. He analyzes the system components to look for potential **Single point of failure** and take appropriate steps to mitigate them. Kris gives special attention to **fault handling** mechansims to ensure that the system is robust and can withstand failures at all levels.

Data Architecture:

Due to the diverse nature of data sources involved and the need for scalability, Kris might have to use **NoSql** databases for certain services. Kris analyzes the read and write patterns, the nature of data operations and the nature of the data to recommend and choose the right NoSql database for the solution. Kris also estimates the sizing of the data storage required and whether the solution will cater to the specified number of Transactions per second.

Security:

Kris evaluates the security requirements of the application and also the organizational compliance laws required to protect customer data privacy. He ensures that application is secured against various attacks like **XSRF, XSS** attacks for example. Kris also makes sure that communication channels are secured by using **SSL** or **MASSL** as required.

Kris generates multiple **view points** to discuss the architecture with various stakeholders.

Note that, all the steps mentioned here need not have been necessarily executed in the same order and some of the steps could be iterated several times, before arriving at the solution. For example, Security requirements would have to be considered in every other step. Similarly, choosing a target platform may not be possible at one step and may have to be revisited as it would be impacted and constrained by other steps like data architecture, application architecture etc.

References

- https://en.wikipedia.org/wiki/ACID
- https://kafka.apache.org/intro.html
- https://www.w3.org/TR/cors/
- https://www.docker.com/what-container
- http://searchservervirtualization.techtarget.com/definition/container-based-virtualization-operating-system-level-virtualization
- http://searchenterprisewan.techtarget.com/definition/chatty-protocol
- http://www.enterpriseintegrationpatterns.com/patterns/messaging/PipesAndFilters.html
- http://whatis.techtarget.com/definition/tower-server
- http://www.dell.com/learn/uk/en/ukbsdt1/sb360/best-server-uk
- http://www.dell.com/uk/business/p/poweredge-t130/pd?c=uk&l=en&s=bsd
- https://blog.dell.com/en-us/design-to-value-yields-customer-s-choice/
- https://en.wikipedia.org/wiki/Solid-state_drive
- http://www.slashroot.in/san-vs-nas-difference-between-storage-area-network-and-network-attached-storage
- http://searchcloudstorage.techtarget.com/feature/How-an-object-store-differs-from-file-and-block-storage
- https://docs.cloudfoundry.org/
- https://en.wikipedia.org/wiki/Content_delivery_network
- 'SOA Principles of Service Design' by Thomas Erl