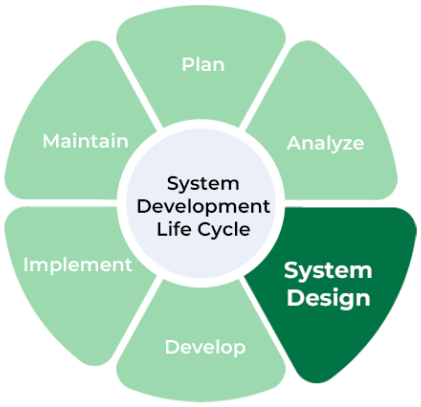
High Level System Design

1.. Introduction

System Design is the core concept behind the design of any distributed systems. System Design is defined as a process of creating an architecture for different components, interfaces, and modules of the system and providing corresponding data helpful in implementing such elements in systems.

In any development process, be it Software or any other tech, the most important stage is Design. Without the designing phase, you cannot jump to the implementation or the testing part. The same is the case with the System as well. System Design not only is a vital step in the development of the system but also provides the backbone to handle exceptional scenarios because it represents the business logic of software.



The importance of System Design phase in SDLC

From the above SDLC steps, it is clear that system design acts as a backbone because no matter how good the coding part is executed, it, later on, becomes irrelevant if the corresponding design is not good. So here we get crucial vital information as to why it is been asked in every Product Based Company.

Objectives of System Design

- **Practicality**:
  We need a system that should be targeting the set of users corresponding to which they are designing.

- **Accuracy and Completeness**
  System which we are designing should meet all the User requirements which is also called Functional Requirement as well as Non-Functional Requirement.

- **Efficient and Optimized**
  The System should be Designed in such a way that it should not overuse/misuse any of the resources. System should give us output response in a fraction of second and should not consume Unnecessary space in Memory.

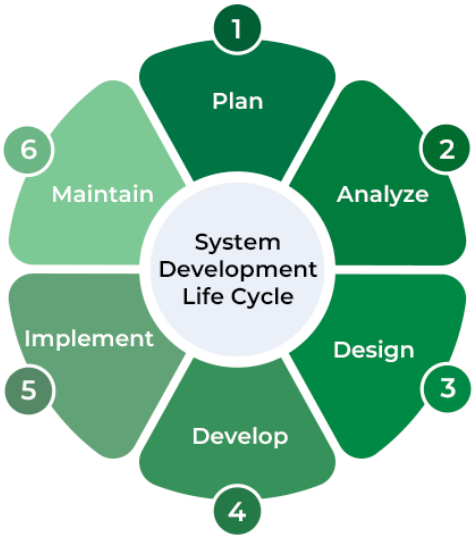
- **Scalable(flexibility):**
  System design should be adaptable with time as per different user needs of customers which we know will keep on changing on time. System should be design in such a way that it can sale up when the traffic increases and application should not be down.

System Development Life Cycle

Any system doesn't get designed in a single day or in a single step. For every system design, there are a series of steps/phases/stages that takes place to get a strong system. This series is defined as System Development Life Cycle (SDLC). We need a strong understanding of understanding life-cycle of a system just likely to define the scope for any variable in code chunk because then only we will be able to grasp deep down how humongous systems are interacting within machines in the real world.

The stages in System Development Life Cycle are as follows:

1. Plan
2. Analyse
3. Design
4. Develop
5. Implement
6. Maintain



System Development Life Cycle

Types of System Design:

System Design takes place in two Phases:

1. High Level Design (HLD)
2. Low Level Design (LLD)

High Level System Design (HLD)

High Level Design in short HLD is the general system design means it refers to the overall system design. It describes the overall description/architecture of the application. It includes the description of system architecture, data base design, brief description on systems, services, platforms and relationship among modules. It is also known as macro level/system design. It is created by solution architect. It converts the Business/client requirement into High Level Solution. It is created first means before Low Level Design.

Low Level System Design (LLD)

Low Level Design in short LLD is like detailing HLD means it refers to component-level design process. It describes detailed description of each and every module means it includes actual logic for every system component and it goes deep into each modules specification. It is also known as micro level/detailed design. It is created by designers and developers. It converts the High Level Solution into Detailed solution. It is created second means after High Level Design.

2.. Types of Applications

Whenever we are designing any Application, we should know which System Architecture we are going to follow to build the Application as per Requirement effectively. When coming to backend of an application, We Strongly create either of the two types of applications:

- Monolithic Application
- Microservices

But talking about Monolithic or Microservices architectures of an application, let's talk about some basic terminologies which we have for an application and servers.

Web Application

A Web application can be developed by using Web Related Technologies like Servlets, JSPs, HTML, CSS, JS etc. Example of Web Application includes Online Library Management, Online Shopping cart etc.

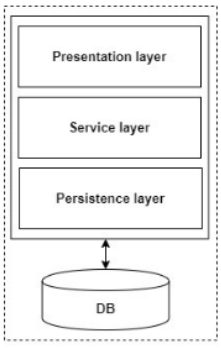
Enterprise Application

An Enterprise Application can be developed by using any technology from Java J2EE which includes all web related Technologies and in addition to that EJBs, JMS components etc. Example – Banking Applications, Telecom Based Project etc. J2EE or JEE compatible applications are Enterprise applications.

Now, we are aware of application types, so we should be good to proceed to know more about Monolithic and Microservices architectures of an Application Design.

Monolithic Architecture

In Monolithic Architecture, complete application exists in one code base. For example, if we want to create an application on e-commerce shopping then the complete shopping application which can be created by using Spring boot, Node JS or Django, and its various services like User Service, Product Service, Order Service, Payment Service exists in a single code base. This is called **Monolithic Architecture**.



Monolithic architecture

There are few disadvantages of Monolithic Architecture:

- If we need to make some changes on any one of the services, then we need to redeploy the complete application.
- Difficult to Scale up individual service.
- Every service will be created in a same technology, we are not flexible enough to select the technology of our choice to build any of the service.
- If there is any error in any of the service, it can create issue in complete application to run.

The advantages of Monolithic Architecture are:

- Single Code Base
- Suitable for small scale applications.
- Easy to develop.

Microservices Architecture

Microservices are created to overcome the disadvantages of Monolithic Architecture. If we want to create Microservices for above example of e-commerce shopping, we can make each of the service as a separate application. So, we will create four applications now:

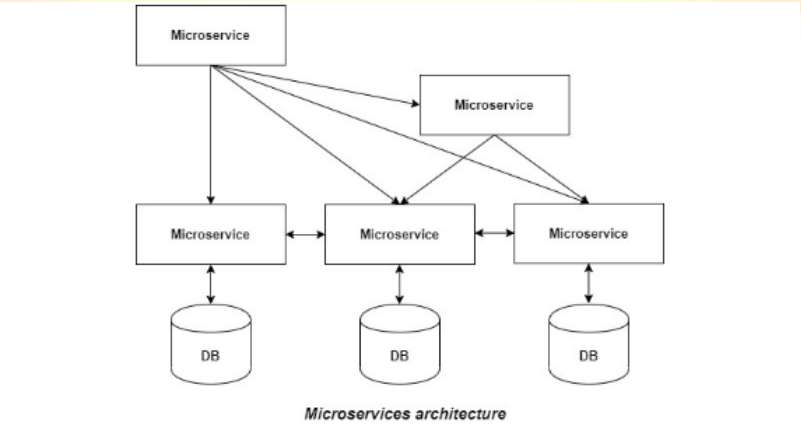
- User Service
- Order Service
- Product Service
- Payment Service

The advantage of designing Microservices is that we can scale each of the service independently without affecting the overall application. And if any changes is required to perform in any of the services we are required to deploy that service alone we are not required to redeploy each of the services.

But Microservices too has disadvantages, if we have not designed microservices properly, then there is a chance of High Latency or Performance problem. Improper microservices design can also lead to tightly coupling between services. Performing Transaction can be difficult as we have multiple services and each service has its own Database.

We have to think how we can decompose our entire Application Design to several Microservices. It can be decomposed based on Business Logic where each service is handling one Business Logic as we did above or we can decompose based on some domains.

Now here the biggest challenge in creating Microservices is how they will communicate and how we will call the services as a single application. This we will discuss later in this documentation.



Microservices architecture

Patterns for Designing Microservices

- Strangler Pattern
- SAGA Pattern
- CQRS Pattern

Strangler Pattern

Strangler Pattern is used if we want to refractor an already existing Monolithic application to Microservices Application. So, What Strangler Pattern Says – "We cannot refractor an entire monolithic Application to Microservices in one Go, we have to do it one by one service. Suppose one of the Service from monolithic is refractor to Microservices then we are not going to divert the 100% traffic to this new Microservice, instead we will just sent 10% traffic to new Microservices

and remaining 90% will be still sent to Old Monolithic Application. If any error is found on Microservices we can make the traffic to Microservices as 0% and complete traffic to Monolithic application. But if no issues found on Microservices, we will slowly increase the % of Traffic on Microservices and decrease the % of Traffic on Monolithic Architecture. At one time Microservices traffic will be reached to 100% and Monolithic Traffic will reached to 0%". This is how Strangler design pattern Works.

SAGA Design Pattern

SAGA pattern says how we can design our Database for Microservices. We can either go for individual database for each Microservice or we can go for Shared Database which can be shared by two or More or all Microservices.

We shouldn't go for Shared Database because say we have three Services A,B and C. Service B generates a data to a very large extent as compared to Service A and C but since they are using shared Database we are required to scale the complete Database. For B this scaling is ok but for A and C there is a waste of money and memory.

We should always go for Individual database for each microservice so that we can scale only the required Database. If any service say A requires the data from database of service B, then service A cannot directly access database of Service B, rather Service A will call the API which will give the data from database of Service B. This is the reason we can go for different Database for each of the services.

As per SAGA design Pattern, The services A, B and C will use the Message Events to complete any transaction and will communicate with the help of same message event in case of any failures and requires a Rollback.

Any Digital Payment application like Net banking, Mastercard, PhonePe will use SAGA Design Pattern for completion of any Payment Transaction.

CQRS Design Pattern

Command Query Request Segregation means Command (Create, Update and Delete) operation should be segregated from Query (Read) Operation. Command operation will be performed on the individual service level Database and Query Operation will be performed on a separate database which serves as a view database. This can be used to perform Join Operation.

Note: A microservices architecture is one type of distributed system, since it decomposes an application into separate components or "services". For example, a microservice architecture may have services that correspond to business features (payments, users, products, etc.) where each corresponding component handles the business logic for that responsibility. The system will then have multiple redundant copies of the services so that there is no central point of failure for a service.

3.. Client Server Architecture

Before jumping directly to Client Server Architecture, let us first know about Client and Server.

Client

A Client is either a person or an organization using a service. In the IT context, the client is a computer/device, also called a Host, that actually uses the service or accepts the information from the server. Client devices include laptops, workstations, IoT devices, and similar network-friendly devices.

Server

A Server in the IT world is a remote computer that provides access to data and services. A server is of two types:

- Web Server
- Application Server

Web Server

Web Servers provides an environment to run Web Application. Web Servers provides support for Web Related Technologies like Servlets, JSPs, HTML, CSS, JS etc. Example of Web Server is Tomcat Web Server.

Application Server

An application server provides an environment to run Enterprise Application. Application servers can provide support for any technologies from Java J2EE which includes all web related Technologies and in addition to that EJBs, JMS components etc. Example of Application Server is Web Logic, JBoss etc. Every application server contains an inbuilt web server to provide support for Web based technologies. If any programmer doesn't want an inbuilt web server, then in that case, he can go for any other web servers. J2EE or JEE compatible server is Application Server.

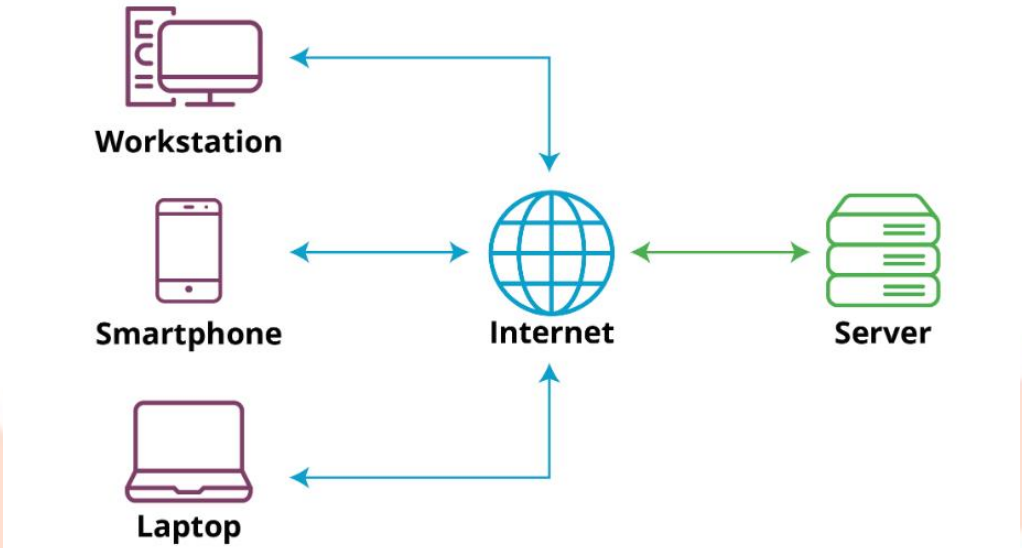
Client Server Architecture

In Client Server Architecture or Client Server model, all requests and services are delivered over a network, and it is also referred to as the **Networking Computing Model** or **Client Server Network**. This architecture breaks down tasks and workloads between clients and servers that reside on the same system or are linked by a computer network.

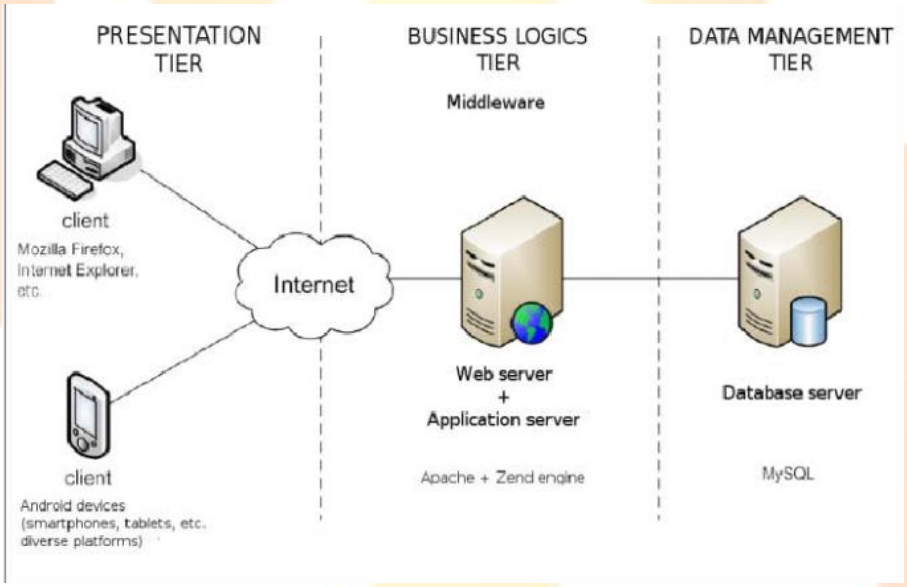
The client sends a request for data, and the server accepts and accommodates the request, process the request and sending the data packets back to the user who needs them.

It is a Unidirectional in nature, where Client is responsible for initiating the Communication. Examples of Protocol which Follows Client Server architecture are SMTP, FTP, HTTP etc. We will learn about these protocols later.

Below is the Basic 2-Tier Client Server Architecture Model. This is called 2-Tier because there is no separate Database Server i.e., Database Server resides in the Server/Application Server.



To create 3-Tier Client Server Architecture Model, we have to separate the Application Server and Database Server. i.e., Client will communicate to the application Server and Application server is responsible to communicate with Database Server and provides the proper and accurate response to the client.



Hence, this is the very first step to Scale any of the Backend application. i.e., Convert 2-Tier Architecture to 3-Tier Architecture. We will discuss Scaling of applications in very detail.

4.. Communication Design Patterns

Till now, we know that Client and Server will Communicate with each other. Client will communicate with the Server to Place a request for any Service and Server has to process the request and communicate back the results to the Client.

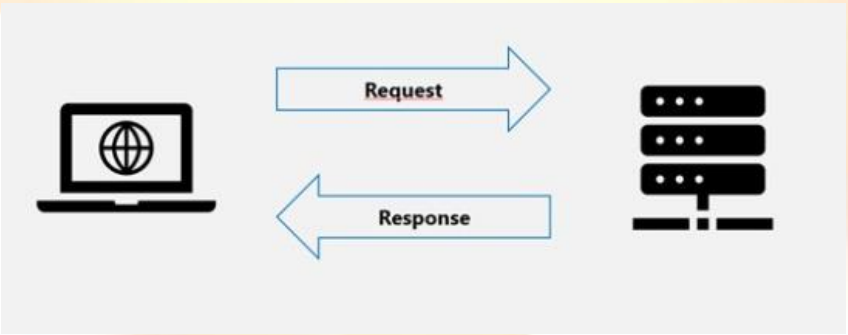
Now, the question arises what communication strategy should Client and Server use to communicate with each other. We have following methodologies with the help of which Client and Server can Communicate with each Other:

- Request & Response
- Synchronous & Asynchronous
- Push
- Polling
- Long Polling
- Server Sent Events
- Publisher & Subscribe
- Multiplexing & Demultiplexing
- Stateful & Stateless
- Sidecar Pattern

4.1 Request & Response

In Request & Response Model, below activities takes place:

1. Client Sends a Request
2. Server Parses the Request
3. Server Processes the Request
4. Server Sends a Response
5. Client Parses the Response and Consume



4.1.2 Where it is Used?

This is the Widely used Communication between Client and Server. The most common use case of Request/Response Model are:

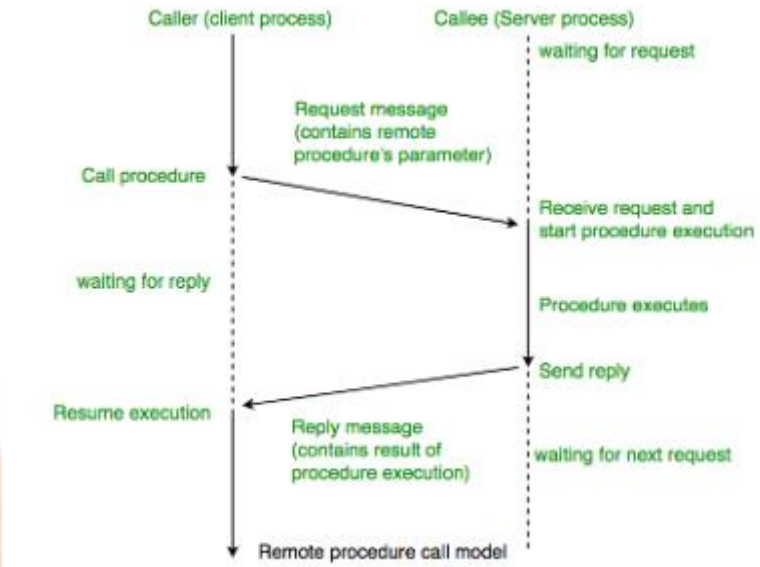
Use – 1: DNS (Domain Name System)

Client which can be Web browser or Mobile application will send a **Request** to DNS (Domain Name System) with the URL either in the Payload or in the Parameter form. DNS will lookup for IP Address of Requested URL and Returns that IP Address to the Client in the form of **Response**.

Use – 2: RPC (Remote Procedure Call)

Remote Procedure Call (RPC) is a powerful technique for constructing distributed, client-server-based applications. It is based on extending the conventional local procedure calling so that the called procedure need not exist in the same address space as the calling procedure. The two

processes may be on the same system, or they may be on different systems with a network connecting them.



When making a Remote Procedure Call:

1. The calling environment is suspended, procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is executed there.
2. When the procedure finishes and produces its results, its results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call.

Use – 3: SQL and Database Protocols

We as a client just hit the SQL query and SQL Server in the backend will parse and process the query and provide us the SQL Result Set in the Response.

Use – 4: APIs (REST/SOAP/GRAPHQL)

API is another Use case of Request/Response Model. Client will hit the URI of an API with Some required Payload as a Request and Server will process that Request and we will get the Response.

4.1.3 Anatomy of the Request/Response

- A Request/Response Structure is Defined by both Client and Server
- Request/Response has a Boundary
- Request/Response is defined by a Protocol and Message Format
- Common Example : HTTP Request

4.1.4 Areas Where we cannot Use Request/Response Pattern

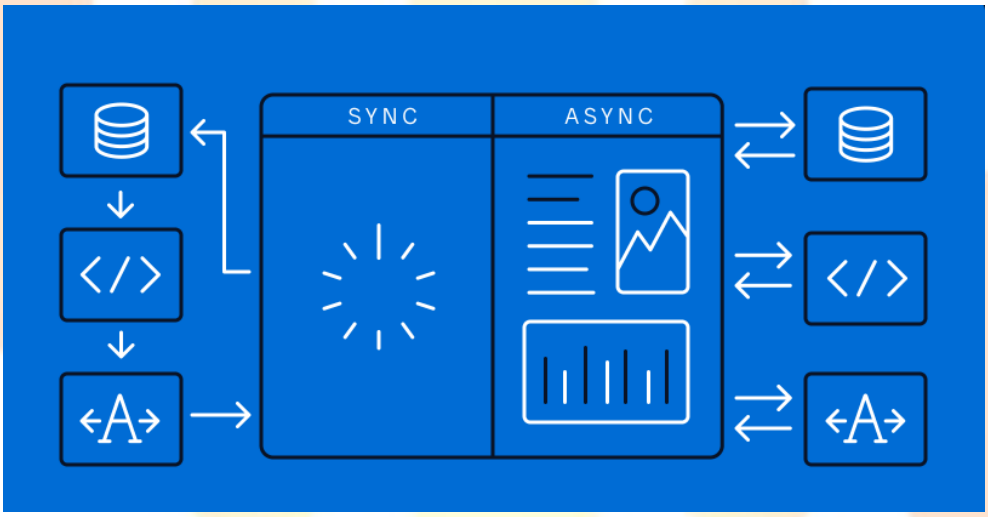
- Notification Service
- Chatting Application

4.2 Synchronous & Asynchronous

4.2.1 Synchronous vs Asynchronous Communication

In **Synchronous Communication**, Client Sends a Request and gets Blocked and cannot execute any code meanwhile. Once Server responds back then only Client gets Unblocked and can execute the remaining code. Here, Client and Server are in "**sync**". Hence, Synchronous is a **blocking** architecture, so the execution of each operation depends on completing the one before it. Each task requires an answer before moving on to the next iteration. When the words Synchronous Communication comes, it means only One Thread can work on any piece of code at a Time. Synchronous Communication is Slower as only one Request gets processed at a Time.

In **Asynchronous Communication**, Client Sends a Request and can work on other piece of Code until gets the Response. Client can either checks if the Response is Ready (epoll) or Server calls back once the Response is Ready (io-uring). Here, Client and Server are not necessary to be in "**sync**". Hence, Asynchronous is a **non-blocking** architecture, so the execution of one task isn't dependent on another. Tasks can run simultaneously. When the words Asynchronous Communication comes, it means Multiple Threads can work on any piece of code at a Time. Asynchronous Communication is faster as multiple Requests gets processed at a Time and hence gives high throughput.



4.2.2 Synchronous vs Asynchronous Programming

Asynchronous programming enhances the user experience by decreasing the lag time between when a function is called and when the value of that function is returned. Async programming translates to a faster, more seamless flow in the real world. For example, users want their apps to run fast, but fetching data from an API takes time. In these cases, asynchronous programming helps the app screen load more quickly, improving the user experience.

Synchronous programming, on the other hand, is advantageous for developers. Synchronous programming is much easier to code. It's well supported among all programming languages, and as the default programming method, developers don't have to spend time learning something new that could open the door to bugs.

4.2.3 Synchronous and Asynchronous Use Cases

Asynchronous programming is critical to programming independent tasks. For instance, asynchronous programs are ideal for development projects with many iterations. Asynchronous programming will keep development moving forward because steps don't have to follow a fixed sequence. Responsive UI is a great use case for asynchronous planning. Take, for example, a shopping app. When a user pulls up their order, the font size should increase. Instead of first

waiting to load the history and update the font size, asynchronous programming can make both actions happen simultaneously.

Asynchronous programming is relatively complex. It can overcomplicate things and make code difficult to read. On the other hand, **synchronous** programming is fairly straightforward; its code is easier to write and doesn't require tracking and measuring process flows (as async does). Because tasks depend on each other, you have to know if they can run independently without interrupting each other. Synchronous programming could also be appropriate for a customer-facing shopping app. Users want to buy all their items together rather than individually when checking out online. Instead of completing an order every time the user adds something to their cart, synchronous programming ensures that the payment method and shipping destination for all items are selected at the same time.

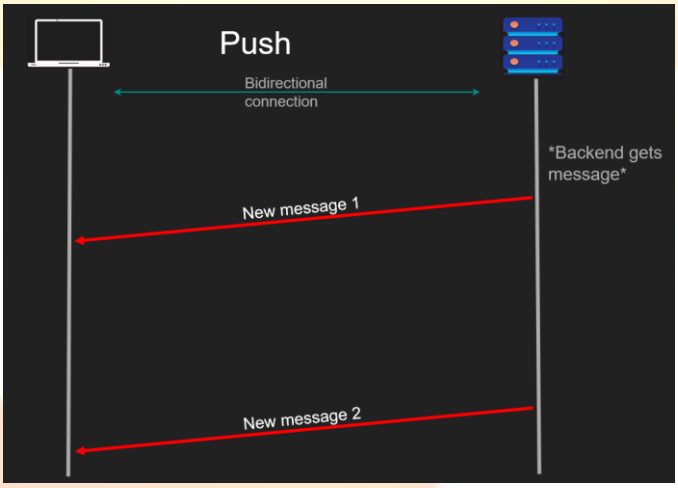
4.3 Push

The strongest use case for push APIs is for instances in which you have time-sensitive data that changes often, and clients need to be updated as soon as that data changes. Push APIs allow the server to send updates to the client whenever new data becomes available, without a need for the client to explicitly request it. When the server receives new information, it pushes the data to the client application, no request needed. This makes it a more efficient communication standard for data that stays changes often.

Hence In **Push** Architecture:

- Client Connects to the Server
- Server sends the data to the client whenever available
- Client doesn't have to do anything i.e., not required to Request Explicitly.

Notification Service is one of the most common Use Case which uses Push Architecture.



The main advantage of Push Notification is We will get the Data from the Server in Real-time once latest data is available to the Server.

The disadvantages of Push Notifications are:

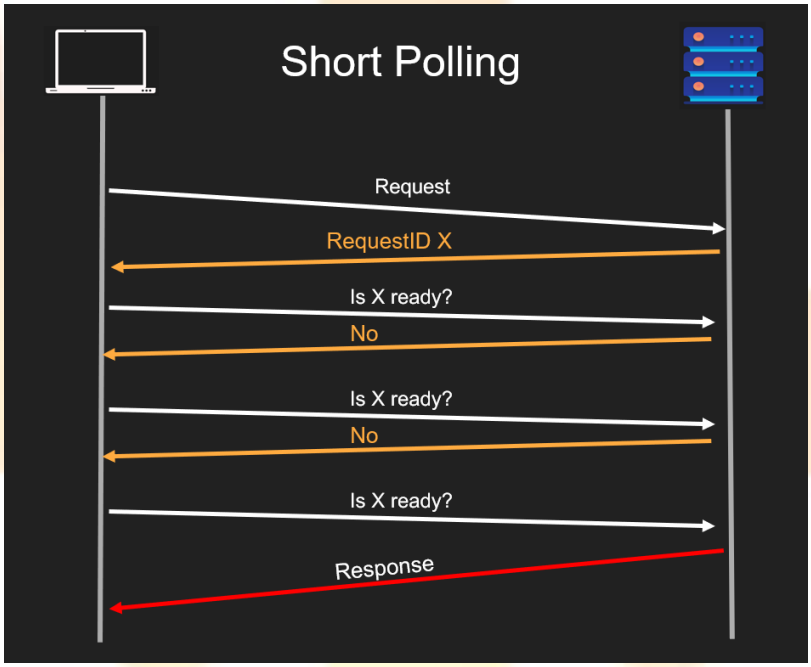
- Clients must be Online i.e., Client should be Connected to the Server.
- Clients has no control
- Protocol required by Push Architecture should be Bidirectional like Web Socket.

4.4 Polling

Polling or Short Polling is also known as Pull Architecture. If your client app needs to find out if there is any new information on the server side, the easiest—if not always the best—way to accomplish this is to call and ask, which is basically how pull APIs work. They are a type of network communication where the client application initiates the communication by requesting updates. The server receives the request, verifies it, processes it, and sends an appropriate response back to the client.

This whole process of polling updates usually takes hundreds of milliseconds. If there are many requests, it can slow down or even overload your server layer, resulting in a poor user experience, customer complaints, and even economic losses. Therefore, to control traffic, many companies often use rate limits to limit how often a client can ask for the same information.

While some of the drawbacks of pull architecture can be mitigated by a system that can distribute processing, even this quickly becomes a very expensive, resource-intensive option. If you need the ability to communicate frequent updates, push architecture is likely a better choice.



Pull vs Push APIs

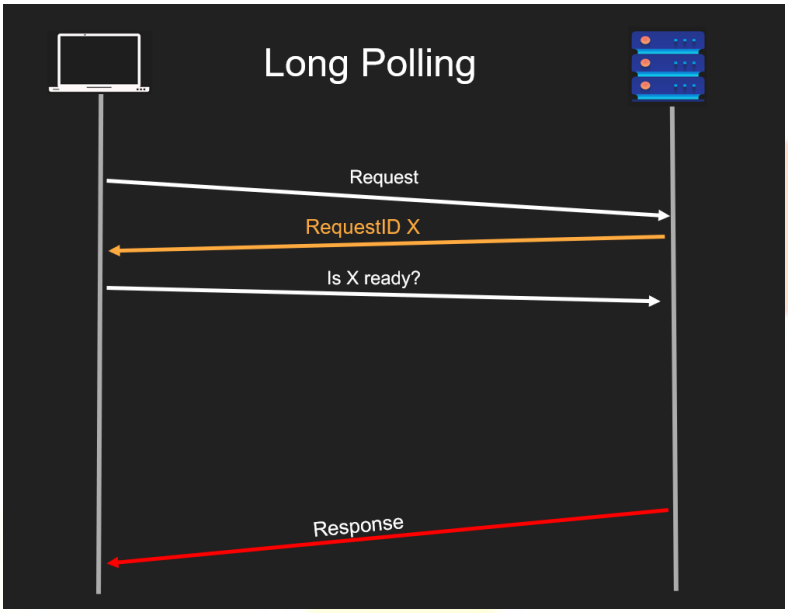
- In a pull API, the client requests the information. In a push API, the server sends the information as it becomes available.
- Pull architecture is request driven: the client sends the request, and the server responds accordingly. Push architecture is event driven: the server pushes data to clients as updates become available.
- In a push API, the server needs to store client details to reach clients directly. With pull, the server doesn’t need to store these details, because they’re encoded in the request.
- Push APIs are significantly faster than pull APIs. In a pull API, the server must receive and verify the request, then process information to form a response that’s sent to the client. In a push API, the server immediately processes information and sends it to clients as soon as it's available.

4.5 Long Polling

In Short polling, there was a problem that if the response is not available then the server returns an empty response. So, In long polling, this problem got solved. Here, in long polling, the client sends a request to the server and if the response is not available then the server will hold the

request till the response gets available, & after the availability of the response, the server will send the response back. After getting a response, again the request will be made either immediately or after some period of time and this process will repeat again and again. In simple words, the client will always be in the live connection to the server.

In Long Polling, a client holds the connection open until there are actually new messages from the server available or a timeout threshold has been reached.

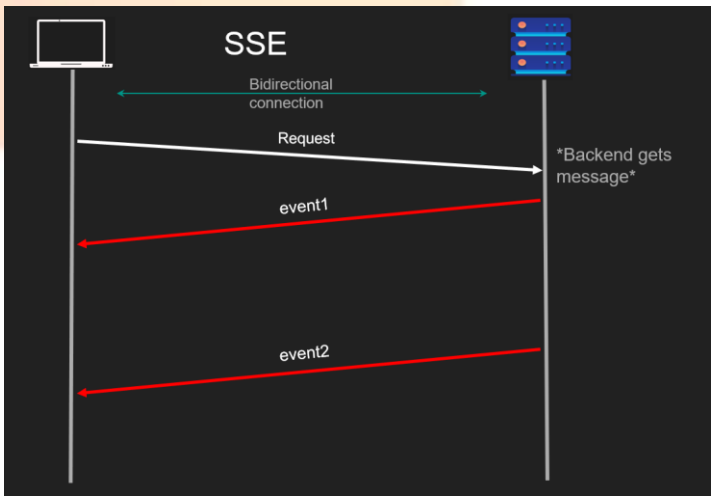


Drawbacks:

- Sender and Receiver may not connect to the same chat server. HTTP Based servers are usually stateless. If we use Round Robin for Load Balancing, the Server that receives the message might not have a long polling connection with the client who receives the message.
- It is inefficient.

4.6 Server Sent Events

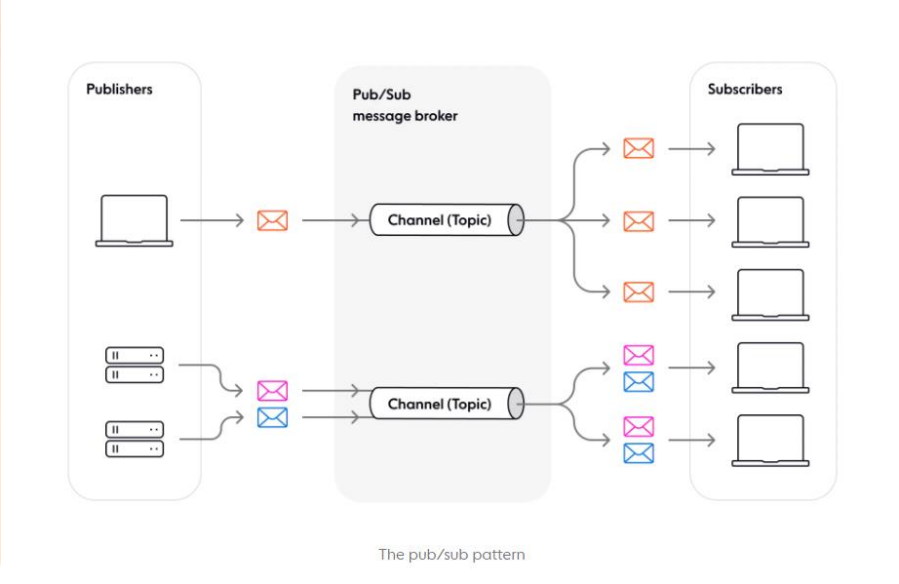
Server-Sent Events (SSE) is a server push technology enabling a client to receive automatic updates from a server via an HTTP connection, and describes how servers can initiate data transmission towards clients once an initial client connection has been established. They are commonly used to send message updates or continuous data streams to a browser client and designed to enhance native, cross-browser streaming through a JavaScript API called Event Source, through which a client requests a particular URL in order to receive an event stream.



4.7 Publish Subscribe Model

Pub/Sub (or Publish/Subscribe) is an architectural design pattern used in distributed systems for asynchronous communication between different components or services. Although Publish/Subscribe is based on earlier design patterns like message queuing and event brokers, it is more flexible and scalable. The key to this is the fact that Pub/Sub enables the movement of messages between different components of the system without the components being aware of each other's identity (they are decoupled).

Pub/Sub provides a framework for exchanging messages between publishers (components that create and send messages) and subscribers (components that receive and consume messages). Note that publishers don't send messages to specific subscribers in a point-to-point manner. Instead, an intermediary is used - a Pub/Sub message broker, which groups messages into entities called channels (or topics).



The pub/sub pattern

The Pub/Sub pattern brings many benefits to the table, including but not limited to:

- Loose coupling between components, making your system more modular and flexible.
- High scalability (in theory, Pub/Sub allows any number of publishers to communicate with any number of subscribers).
- Language-agnostic and protocol-agnostic, which makes it straightforward and fast to integrate Pub/Sub into your tech stack.
- Asynchronous, event-driven communication that's ideal for real time, low-latency apps.

Use Cases:

Pub/Sub's loose coupling, asynchronous nature, and inherent scalability make it an excellent solution for distributed systems with a high and fluctuating number of publishers and subscribers. You can use Pub/Sub for many different purposes, such as:

- Sending event notifications.
- Distributed caching.
- Distributed logging.
- Working with multiple data sources.
- Broadcasting updates (one-to-many messaging).
- Building responsive, low-latency end-user experiences like live chat and multiplayer collaboration functionality.

4.8 Multiplexing vs Demultiplexing

Multiplexing and Demultiplexing services are provided in almost every protocol architecture ever designed. UDP and TCP perform the demultiplexing and multiplexing jobs by including two special fields in the segment headers: the source port number field and the destination port number field.

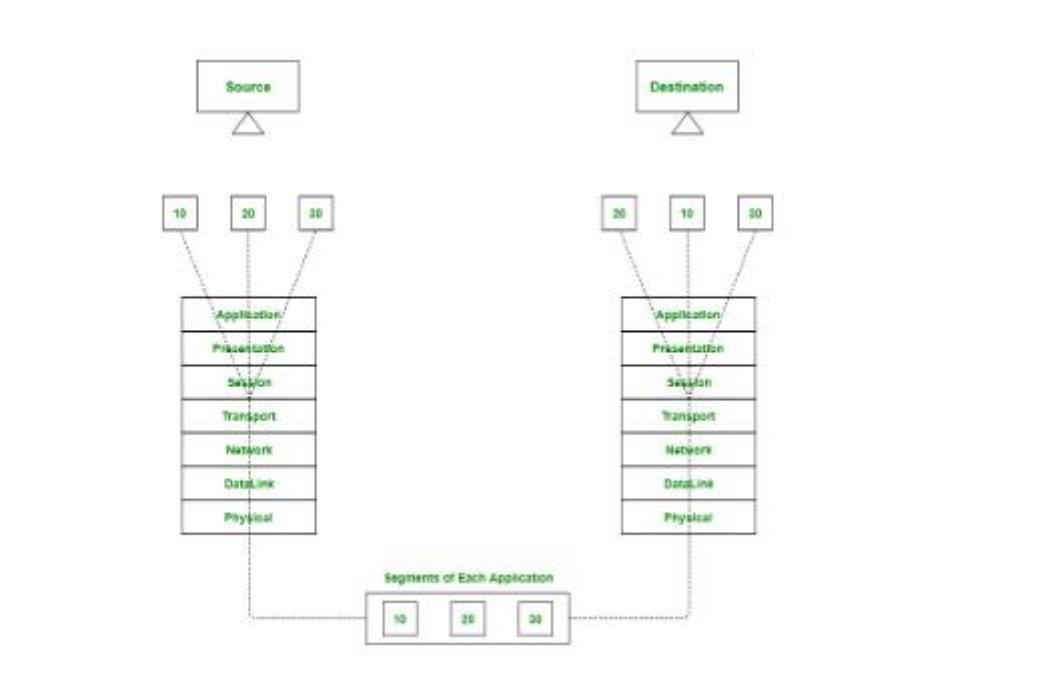
Multiplexing

Gathering data from multiple application processes of the sender, enveloping that data with a header, and sending them as a whole to the intended receiver is called multiplexing.

Demultiplexing

Delivering received segments at the receiver side to the correct app layer processes is called demultiplexing.

Multiplexing and demultiplexing are the services facilitated by the transport layer of the OSI model.

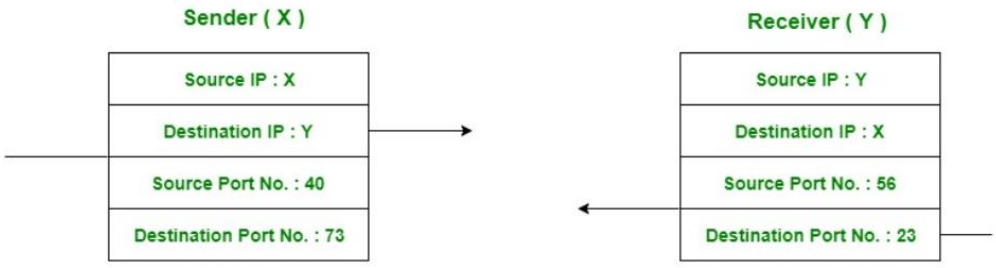


There are two types of multiplexing and Demultiplexing:

- Connectionless Multiplexing and Demultiplexing
- Connection-Oriented Multiplexing and Demultiplexing

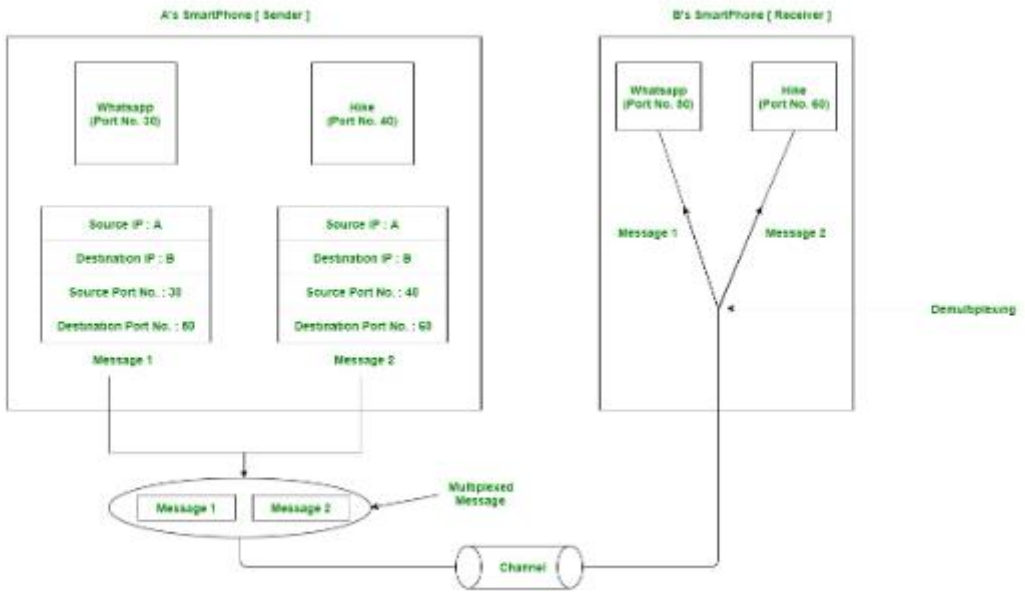
How Multiplexing and Demultiplexing is done?

For sending data from an application on the sender side to an application at the destination side, the sender must know the IP address of the destination and port number of the application (at the destination side) to which he wants to transfer the data. Block diagram is shown below:



Let us consider two messaging apps that are widely used nowadays viz. Hike and WhatsApp. Suppose A is the sender and B is the receiver. Both sender and receiver have these applications installed in their system (say smartphone). Suppose A wants to send messages to B in WhatsApp and hike both. In order to do so, A must mention the IP address of B and destination port number of the WhatsApp while sending the message through the WhatsApp application. Similarly, for the latter case, A must mention the IP address of B and the destination port number of the hike while sending the message.

Now the messages from both the apps will be wrapped up along with appropriate headers(viz. source IP address, destination IP address, source port no, destination port number) and sent as a single message to the receiver. This process is called multiplexing. At the destination, the received message is unwrapped and constituent messages (viz messages from a hike and WhatsApp application) are sent to the appropriate application by looking to the destination the port number. This process is called demultiplexing. Similarly, B can also transfer the messages to A.



4.9 Stateful & Stateless

In programming, "state" refers to the condition of a system, component, or application at a particular point in time.

As a simple example, if you are shopping on amazon.com, whether you are currently logged into the site or if you have anything stored in your cart are some examples of state. State represents the data that is stored and used to keep track of the current status of the application. Understanding and managing state is crucial for building interactive and dynamic web applications.

The concept of a “state” crosses many boundaries in architecture. Design patterns (like REST and GraphQL), protocols (like HTTP and TCP), firewalls and functions can be stateful or stateless. But the underlying principle of “state” cutting across all of these domains remains the same. This article will explain what state means. It will also explain stateful and stateless architectures with some analogies and the benefits and trade-offs of both.

4.9.1 What is Stateful Architecture?

Imagine you go to a pizza restaurant to eat some food. In this restaurant, there is only a single waiter, and the waiter takes detailed notes on your table number, what you ordered, your

preferences based on past orders, like what type of pizza crust you like or toppings you are allergic to, and so on.

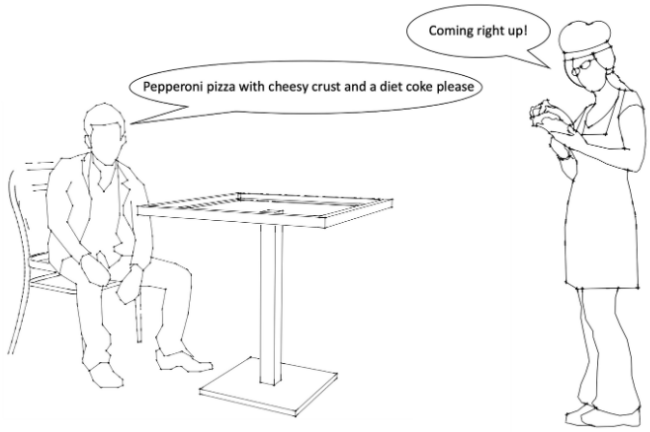
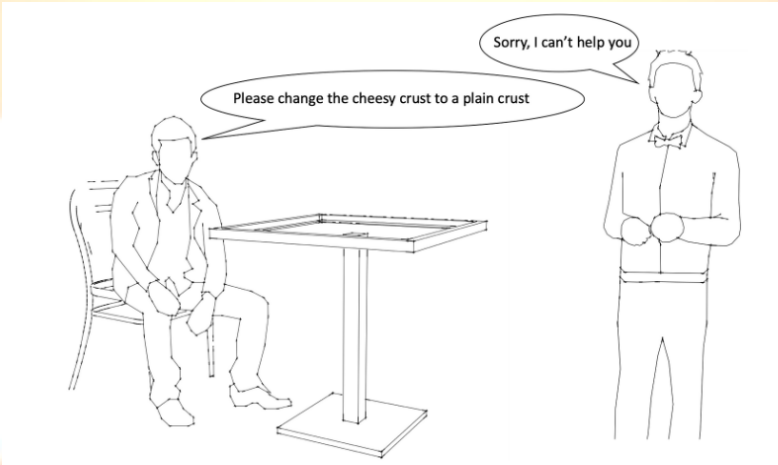


Illustration of a waiter taking a person's order at a pizza restaurant

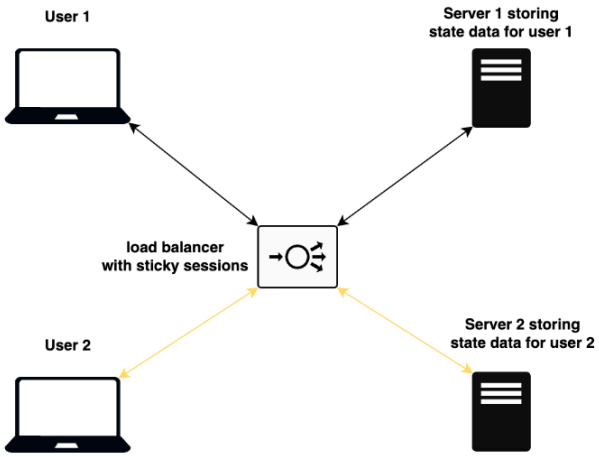
All of these pieces of information that the waiter writes down in their notepad are the customer's state. Only the waiter serving you has access to this information. If you want to make a change to your order or check how it's coming along, you need to speak to the same waiter that took your order. But since there is only one waiter, that is not a problem.

Now, suppose the restaurant starts to get busier. Your waiter has to respond to other guests so more waiters are called to work. You now want to check the status of your order and make a small change to it – a plain crust instead of a cheesy crust. The only available waiter is different from the one who initially took your order.



This new waiter does not have details of your order, that is your state. Naturally, they will not be able to check the status of your order or make changes to it. A restaurant that operates like this, where only the waiter that initially took your order can give you updates about it, or make changes to it, follows a stateful design.

Similarly, a stateful application will have a server that remembers clients' data (that is, their state). All future requests will be routed to the same server using a load balancer with sticky sessions enabled. In this way, the server is always aware of the client. The diagram below shows two different users trying to access a web server through a load balancer. Since the application state is maintained on the servers, the users must always be routed to the same server for every single request in order to preserve state.



Sticky sessions are a configuration that allows the load balancer to route a user's requests consistently to the same backend server for the duration of their session. This is in contrast to traditional load balancing, where requests from a user can be directed to any available backend server in a round-robin or other load distribution pattern.

What is the problem with a stateful architecture? Imagine a restaurant run in this manner. While it may be ideal and easy to implement for a small, family run restaurant with only a few customers, such a design is not fault tolerant and not scalable.

What happens if the waiter who took a customer's order has an emergency and needs to leave? All the information regarding that order leaves with that waiter as well. This disrupts the customer’s experience, since any new waiter brought in to replace the old one has no knowledge of previous orders. This is a design that is not fault tolerant.

Also, having to distribute requests so that the same customer can only speak to the same waiter means that the load on different waiters is not equally distributed. Some waiters will be overwhelmed with requests if you have a very demanding customer who always modifies or adds things to their order. Some of the other waiters will have nothing to do, and can’t step in to help. Again, this is a non-scalable design.

Similarly, storing state data for different customers on different servers is not fault tolerant and not scalable. A server failure will lead to loss of state data. So, if a user is logged in and about to check out for a large order on Amazon.com for example, the user will be forced to re-authenticate and the user's basket will be empty. They would have to log in again and fill up their basket from scratch – a poor user experience.

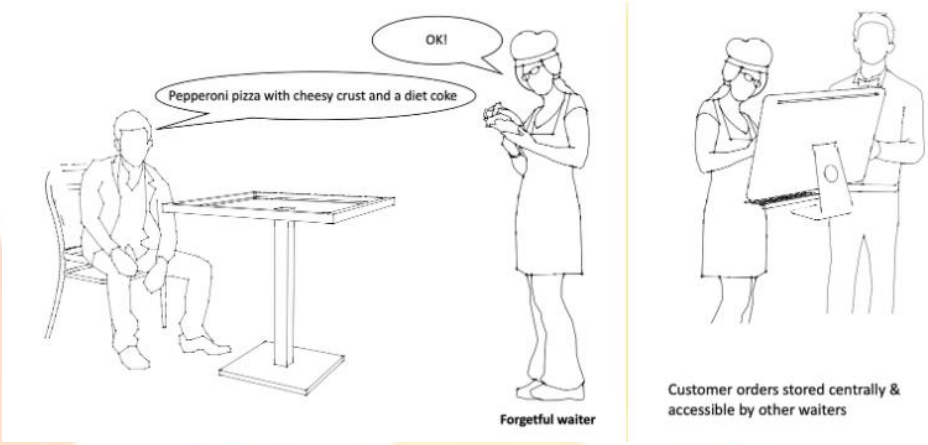
Scalability will also be difficult to achieve during peak times like Black Friday with a stateful design. New servers will be added to the auto scaling group but since sticky sessions are enabled, clients will be routed to the same server, causing them to be overwhelmed, which can cause an increase in response times - a poor user experience.

Stateless architectures solve a lot of these problems.

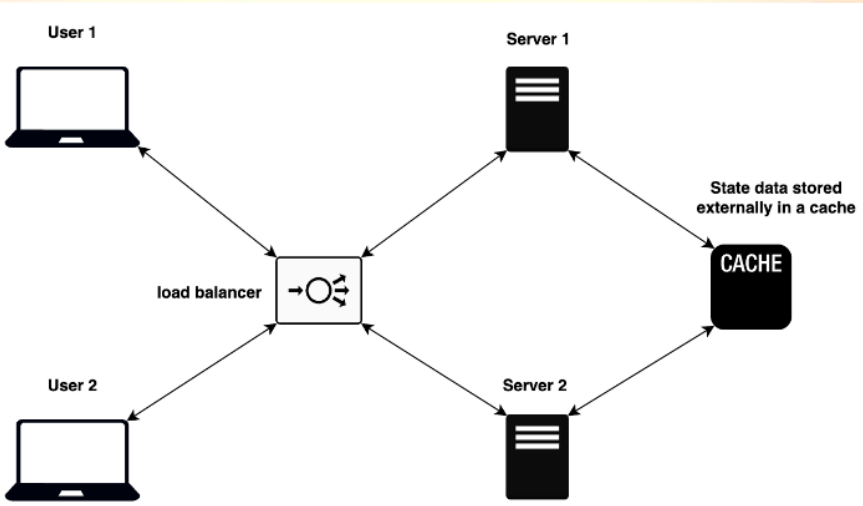
4.9.2 What is Stateless Architecture?

“Stateless” architecture is a confusing term, as it implies the system is without state. A stateless architecture does not, however, mean that state information is not stored. It simply means that state information is stored outside of the server. Therefore, the state of being stateless only applies to the server.

Bringing back the restaurant analogy, waiters in a stateless restaurant can be thought of as having perfectly forgetful memories. They do not recognise old customers, and can't recall what you ordered or how you like your pizza. They will simply take note of customers' orders on a separate system, say a computer, that is accessible by all the waiters. They can then revert back to the computer to get details of an order and make changes to it as required.



By storing the 'state' of a customer's order on a central system accessible by other waiters, any waiter can serve any customer. In a stateless architecture, HTTP requests from a client can be sent to any of the servers. State is typically stored on a separate database, accessible by all the servers. This creates a fault tolerant and scalable architecture since web servers can be added or removed as needed, without impacting state data. The load will also be equally distributed across all servers, since the load balancer will not need a sticky session configuration to route the same clients to the same servers. The diagram below shows two different users trying to access a web server through a load balancer. Since the application state is maintained separately from the servers, the users can be routed to any of the servers, which will then get the state information from an external database accessible by both servers.



Typically, state data is stored in a cache like Redis, an in-memory data store. Storing state data in-memory improves read and write times, compared to storing it on disk, as explained here.

If we look at network protocols as an example, **HTTP** is a stateless protocol. This means that each HTTP request from a client to a server is independent and carries no knowledge of previous requests or their context. The server treats each request as a separate and isolated transaction, and it doesn't inherently maintain information about the state of the client between requests.

State is either maintained on the servers (stateful architecture) or in a separate database outside the servers (stateless architecture). The HTTP protocol itself does not maintain state.

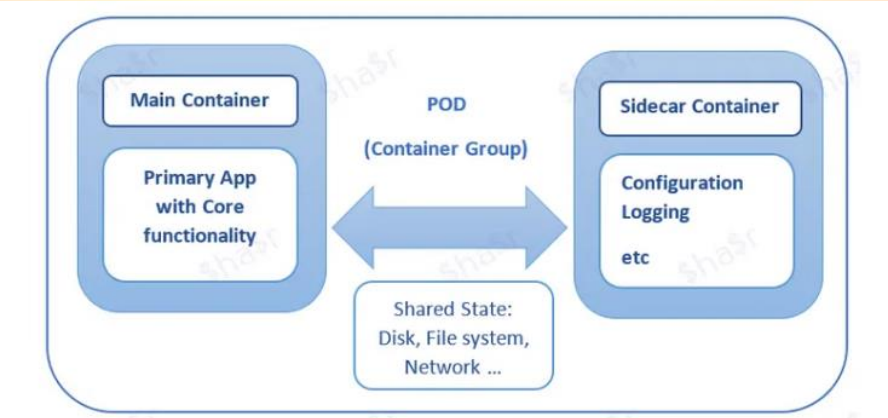
Unlike the stateless nature of HTTP, the **TCP** protocol is connection-oriented and stateful. It establishes a connection between two devices (usually a client and a server) and maintains a continuous communication channel until the connection is terminated.

The key principle behind something that is stateful is that it has perfect memory or knowledge of previous calls or requests, while something that is stateless has no memory or knowledge of previous calls or requests.

4.10 Sidecar Pattern

Sidecar concept in Kubernetes is getting popular and. It's a common principle in container world, that container should address a single concern & it should do it well. The Sidecar pattern achieves this principle by decoupling the core business logic from additional tasks that extends the original functionality. Sidecar pattern is a single node pattern made up of two containers. The first is the application container which contains the core logic of the application (primary application). Without this container, application wouldn't exist.

In addition, there is a Sidecar container used to extend/enhance the functionalities of the primary application by running another container in parallel on the same container group (Pod). Since sidecar runs on the same Pod as the main application container it shares the resources — filesystem, disk, network etc., It also allows the deployment of components (implemented with different technologies) of the same application into a separate, isolated & encapsulated containers. It proves extremely useful when there is an advantage to share the common components across the microservice architecture (e.g.: logging, monitoring, configuration properties etc.)



Eg: Sidecar: Pod with 2 containers

Examples:

- Adding HTTPS to a Legacy Service
- Dynamic Configuration with Sidecars
- Log Aggregator with Sidecar

5.. Protocols

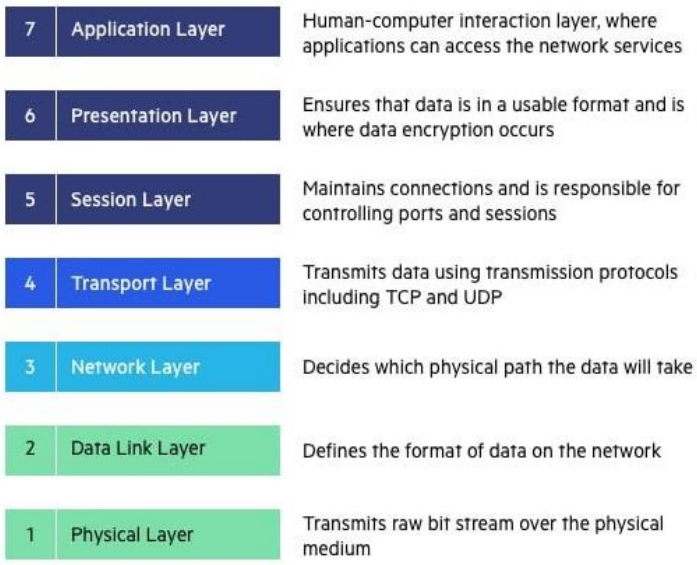
Now, since we are aware of various Communication Patterns with which Client and Server is going to Communicate. Now we will discuss about Various Protocols through which these Communications will takes Place.

5.1 OSI Model

The Open Systems Interconnection (OSI) model describes seven layers that computer systems use to communicate over a network. It was the first standard model for network communications, adopted by all major computer and telecommunication companies in the early 1980s

The modern Internet is not based on OSI, but on the simpler TCP/IP model. However, the OSI 7-layer model is still widely used, as it helps visualize and communicate how networks operate, and helps isolate and troubleshoot networking problems.

OSI was introduced in 1983 by representatives of the major computer and telecom companies, and was adopted by ISO as an international standard in 1984.



We'll describe OSI layers "top down" from the application layer that directly serves the end user, down to the physical layer.

7. Application Layer

The application layer is used by end-user software such as web browsers and email clients. It provides protocols that allow software to send and receive information and present meaningful data to users. A few examples of application layer protocols are the Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), Post Office Protocol (POP), Simple Mail Transfer Protocol (SMTP), and Domain Name System (DNS). Example: When you use a web browser to access a website, also the application layer handles the HTTP request and response, enabling you to view and interact with the site's content.

6. Presentation Layer

The presentation layer prepares data for the application layer. It defines how two devices should encode, encrypt, and compress data so it is received correctly on the other end. The presentation layer takes any data transmitted by the application layer and prepares it for transmission over the session layer. Example: When you send a JSON payload from a frontend application to a backend

API, the presentation layer ensures that the data is encoded and decoded correctly to maintain compatibility.

5. Session Layer

The session layer creates communication channels, called sessions, between devices. It is responsible for opening sessions, ensuring they remain open and functional while data is being transferred, and closing them when communication ends. The session layer can also set checkpoints during a data transfer—if the session is interrupted, devices can resume data transfer from the last checkpoint.

Example: When you log in to an online chat application, the session layer sets up a session between your device and the chat server, enabling real-time communication.

4. Transport Layer

The transport layer takes data transferred in the session layer and breaks it into "segments" on the transmitting end. It is responsible for reassembling the segments on the receiving end, turning it back into data that can be used by the session layer. The transport layer carries out flow control, sending data at a rate that matches the connection speed of the receiving device, and error control, checking if data was received incorrectly and if not, requesting it again.

Example: When you stream a video from a media server, the transport layer ensures that the video data is delivered in the correct order and retransmits any lost packets to maintain smooth playback.

3. Network Layer

The network layer has two main functions. One is breaking up segments into network packets, and reassembling the packets on the receiving end. The other is routing packets by discovering the best path across a physical network. The network layer uses network addresses (typically Internet Protocol addresses) to route packets to a destination node.

Example: When you access a website hosted on a remote server, the network layer helps route your request from your local network through various routers and switches on the internet to reach the server.

2. Data Link Layer

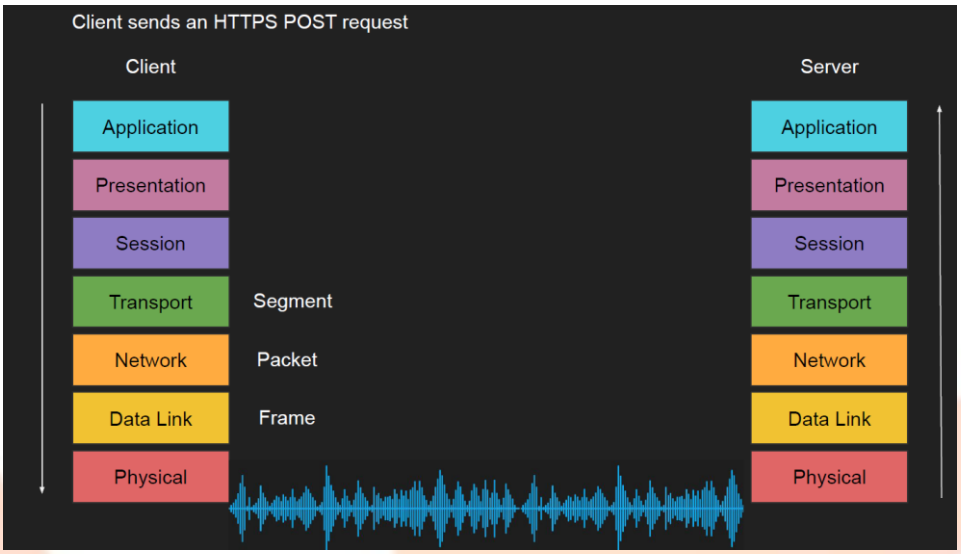
The data link layer establishes and terminates a connection between two physically-connected nodes on a network. It breaks up packets into frames and sends them from source to destination. This layer is composed of two parts—Logical Link Control (LLC), which identifies network protocols, performs error checking and synchronizes frames, and Media Access Control (MAC) which uses MAC addresses to connect devices and define permissions to transmit and receive data.

Example: When you upload a file to a cloud storage service, the data link layer ensures that the data packets are transmitted correctly, detecting and retransmitting any lost or corrupted packets.

1. Physical Layer

The physical layer is responsible for the physical cable or wireless connection between network nodes. It defines the connector, the electrical cable or wireless technology connecting the devices, and is responsible for transmission of the raw data, which is simply a series of 0s and 1s, while taking care of bit rate control.

Example: When you send an HTTP request to a web server, the physical layer is involved in transmitting the request data through the network cables to the server.



5.2 TCP/IP Model

There are four layers of the TCP/IP model: network access, internet, transport, and application. Used together, these layers are a suite of protocols. The TCP/IP model passes data through these layers in a particular order when a user sends information, and then again in reverse order when the data is received.

Layer 1: Network Access Layer

The network access layer, also known as the data link layer, handles the physical infrastructure that lets computers communicate with one another over the internet. This covers ethernet cables, wireless networks, network interface cards, device drivers in your computer, and so on. The network access layer also includes the technical infrastructure — such as the code that converts digital data into transmittable signals — that makes network connection possible.

Layer 2: Internet Layer

The internet layer, also known as the network layer, controls the flow and routing of traffic to ensure data is sent speedily and accurately. This layer is also responsible for reassembling the data packet at its destination. If there's lots of internet traffic, the internet layer may take a little longer to send a file, but there will be a smaller chance of an error corrupting that file.

Layer 3: Transport Layer

The transport layer provides a reliable data connection between two communicating devices. It's like sending an insured package: The transport layer divides the data in packets, acknowledges the packets it has received from the sender, and ensures that the recipient acknowledges the packets it receives.

Layer 4: Application Layer

The application layer is the group of applications that let the user access the network. For most of us that means email, messaging apps, and cloud storage programs. This is what the end-user sees and interacts with when sending and receiving data.

Which IP addresses do TCP/IP work with?

Whether you have an IPv4 or IPv6 address, you're likely already using the TCP/IP model. This is the standard model for most existing internet infrastructure. There are different categories of IP

addresses that may affect your privacy or how the protocol works — such as public vs. local IP addresses or static vs. dynamic IPs — but they still follow the standard TCP/IP model.

TCP/IP: The most common protocol

TCP/IP is the most commonly used protocol suite on the web — so common that most people don't even realize they're using it. Most computers have TCP/IP built in as standard, so there's no manual setup required. Just connect to your local wireless network, and you're ready to go. But while TCP/IP is the most common protocol, it's not the most secure. That's why our security experts built AVG Secure VPN, which provides strong encryption on the OpenVPN protocol (in Windows and Android) and the IKEv2 protocol (for Mac OS and iOS devices) — both of which are more secure than the standard TCP or IP protocols.

5.3 IP (Internet Protocol)

Internet Protocol (IP) is the method or protocol by which data is sent from one computer to another on the internet. Each computer known as a host on the internet has at least one IP address that uniquely identifies it from all other computers on the internet. IP is the defining set of protocols that enable the modern internet. It was initially defined in May 1974 in a paper titled, "A Protocol for Packet Network Intercommunication," published by the Institute of Electrical and Electronics Engineers and authored by Vinton Cerf and Robert Kahn. At the core of what is commonly referred to as IP are additional transport protocols that enable the actual communication between different hosts. One of the core protocols that runs on top of IP is the Transmission Control Protocol (TCP), which is often why IP is also referred to as TCP/IP. However, TCP isn't the only protocol that is part of IP.

5.3.1 How does IP routing work?

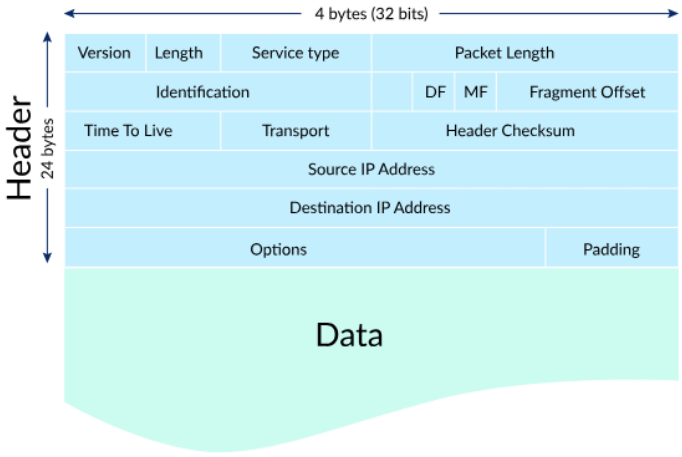
When data is received or sent such as an email or a webpage the message is divided into chunks called **Packets**. Each packet contains both the sender's IP address and the receiver's IP address. Any packet is sent first to a gateway computer that understands a small part of the internet. The gateway computer reads the destination address and forwards the packet to an adjacent gateway that in turn reads the destination address and so forth until one gateway recognizes the packet as belonging to a computer within its immediate neighbourhood or domain. That gateway then forwards the packet directly to the computer whose address is specified.

Because a message is divided into a number of packets, each packet can, if necessary, be sent by a different route across the internet. Packets can arrive in a different order than the order they were sent. The Internet Protocol just delivers them. It's up to another protocol the Transmission Control Protocol to put them back in the right order.

5.3.2 IP packets

While IP defines the protocol by which data moves around the internet, the unit that does the actual moving is the **IP packet**. An **IP Packet** is like a physical parcel or a letter with an envelope indicating address information and the data contained within. An IP packet's envelope is called the **Header**. The packet header provides the information needed to route the packet to its destination. An IP packet header is up to 24 bytes long and includes the **Source IP address**, the **Destination IP address** and Information about the Size of the Whole Packet.

The other key part of an IP packet is the Data Component, which can vary in size. Data inside an IP packet is the content that is being transmitted.



5.3.3 What is an IP address?

IP provides mechanisms that enable different systems to connect to each other to transfer data. Identifying each machine in an IP network is enabled with an IP address. Similar to the way a street address identifies the location of a home or business, an IP address provides an address that identifies a specific system so data can be sent to it or received from it.

An IP address is typically assigned via the **DHCP (Dynamic Host Configuration Protocol)**. DHCP can be run at an internet service provider, which will assign a public IP address to a particular device. A public IP address is one that is accessible via the public internet. A local IP address can be generated via DHCP running on a local network router, providing an address that can only be accessed by users on the same local area network.

5.3.4 Differences between IPv4 and IPv6

The most widely used version of IP for most of the internet's existence has been Internet Protocol Version 4 (IPv4). IPv4 provides a 32-bit IP addressing system that has four sections. For example, a sample IPv4 address might look like 192.168.0.1, which coincidentally is also commonly the default IPv4 address for a consumer router. IPv4 supports a total of 4,294,967,296 addresses.

A key benefit of IPv4 is its ease of deployment and its ubiquity, so it is the default protocol. A drawback of IPv4 is the limited address space and a problem commonly referred to as IPv4 address exhaustion. There aren't enough IPv4 addresses available for all IP use cases. Since 2011, **IANA (Internet Assigned Numbers Authority)** hasn't had any new IPv4 address blocks to allocate. As such, Regional Internet Registries (RIRs) have had limited ability to provide new public IPv4 addresses.

In contrast, IPv6 defines a 128-bit address space, which provides substantially more space than IPv4, with 340 trillion IP addresses. An IPv6 address has eight sections. The text form of the IPv6 address is xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx, where each x is a hexadecimal digit, representing 4 bits. The massive availability of address space is the primary benefit of IPv6 and its most obvious impact. The challenges of IPv6, however, are that it is complex due to its large address space and is often challenging for network administrators to monitor and manage.

5.3.5 IP network protocols

IP is a connectionless protocol, which means that there is no continuing connection between the end points that are communicating. Each packet that travels through the internet is treated as an independent unit of data without any relation to any other unit of data. The reason the packets are reassembled in the right order is because of TCP, the connection-oriented protocol that keeps track of the packet sequence in a message. In the OSI model (Open Systems Interconnection), IP

is in layer 3, the networking layer. There are several commonly used network protocols that run on top of IP, including:

- Transmission Control Protocol (TCP) enables the flow of data across IP address connections.
- User Datagram Protocol (UDP) provides a way to transfer low-latency process communication that is widely used on the internet for DNS lookup and voice over Internet Protocol.
- File Transfer Protocol (FTP) is a specification that is purpose-built for accessing, managing, loading, copying and deleting files across connected IP hosts.
- Hypertext Transfer Protocol (HTTP) is the specification that enables the modern web. HTTP enables websites and web browsers to view content. It typically runs over port 80.
- Hypertext Transfer Protocol Secure (HTTPS) is HTTP that runs with encryption via Secure Sockets Layer or Transport Layer Security (TLS). HTTPS typically is served over port 443.

5.3.6 Problem with Internet Protocols

The Internet Protocol (IP) describes how to split messages into multiple IP packets and route packets to their destination by hopping from router to router. IP does not handle all the consequences of packets, however. For example:

- A computer might send multiple messages to a destination, and the destination needs to identify which packets belong to which message.
- Packets can arrive out of order. That can happen especially if two packets follow different paths to the destination.
- Packets can be corrupted, which means that for some reason, the received data no longer matches the originally sent data.
- Packets can be lost due to problems in the physical layer or in routers' forwarding tables. If even one packet of a message is lost, it may be impossible to put the message back together in a way that makes sense.
- Similarly, packets might be duplicated due to accidental retransmission of the same packet.

Fortunately, there are higher level protocols in the Internet protocol stack that can deal with these problems. **Transmission Control Protocol (TCP)** is the data transport protocol that's most commonly used on top of IP and it includes strategies for packet ordering, retransmission, and data integrity. **User Datagram Protocol (UDP)** is an alternative protocol that solves fewer problems but offers faster data transport.

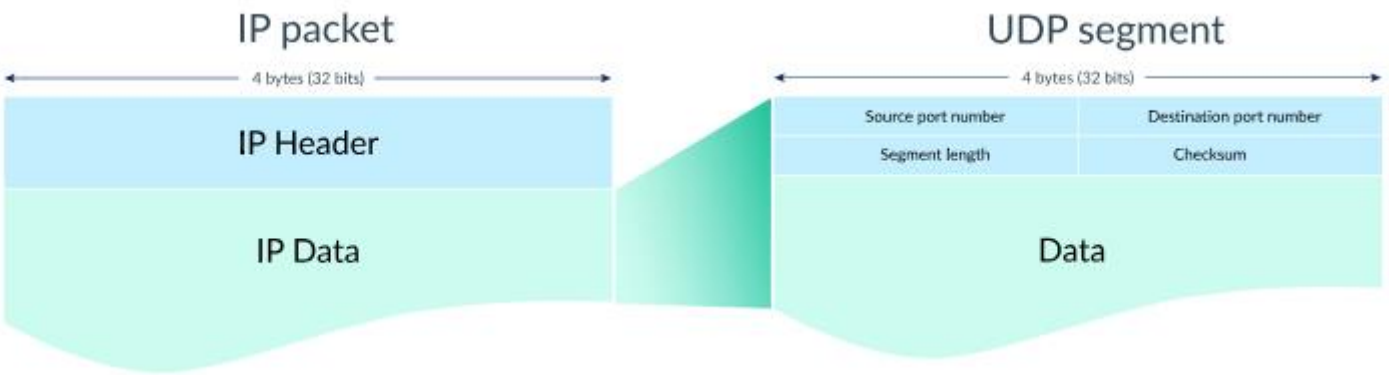
Internet applications can choose the data transport protocol that makes the most sense for their application.

5.4 UDP (User Datagram Protocol)

The User Datagram Protocol (UDP) is a lightweight data transport protocol that works on top of IP. UDP provides a mechanism to detect corrupt data in packets, but it does not attempt to solve other problems that arise with packets, such as lost or out of order packets. That's why UDP is sometimes known as the Unreliable Data Protocol. UDP is simple but fast, at least in comparison to other protocols that work over IP. It's often used for time-sensitive applications (such as real-time video streaming) where speed is more important than accuracy.

5.4.1 Packet format

When sending packets using UDP over IP, the data portion of each IP packet is formatted as a UDP segment.



Each UDP segment contains an 8-byte header and variable length data.

Port Numbers

The first four bytes of the UDP header store the port numbers for the source and destination. A networked device can receive messages on different virtual ports, similar to how an ocean harbour can receive boats on different ports. The different ports help distinguish different types of network traffic.

```
$ sudo lsof -i -n -P | grep UDP
launchd            1   IPv4    UDP *:137
launchd            1   IPv4    UDP *:138
syslogd           45   IPv4    UDP *:54465
mDNSResponder    186   IPv4    UDP *:5353
mDNSResponder    186   IPv6    UDP *:5353
mDNSResponder    186   IPv4    UDP *:65327
mDNSResponder    186   IPv6    UDP *:65327
mDNSResponder    186   IPv4    UDP *:55657
mDNSResponder    186   IPv6    UDP *:55657
Google         12306   IPv6    UDP *:5353
```

Each row starts with the name of the process that's using the port and ends with the protocol and port number.

Segment Length

The next two bytes of the UDP header store the length (in bytes) of the segment (including the header).

Two bytes is 16 bits, so the length can be as high as this binary number: 1111111111111111. In decimal, that's (2^{16}-1) or 65,535. Thus, the maximum length of a UDP segment is 65,535 bytes.

Checksum

The final two bytes of the UDP header is the checksum, a field that's used by the sender and receiver to check for data corruption.

Before sending off the segment, the sender:

- Computes the checksum based on the data in the segment.
- Stores the computed checksum in the field.

Upon receiving the segment, the recipient:

- Computes the checksum based on the received segment.

- Compares the checksums to each other. If the checksums aren't equal, it knows the data was corrupted.

To understand how a checksum can detect corrupted data, let's follow the process to compute a checksum for a very short string of data: "Hola".

First, the sender would encode "Hola" into binary somehow. The following encoding uses the ASCII/UTF-8 encoding:

H	o	l	a
01001000	01101111	01101100	01100001

That encoding gives these 4 bytes:

01001000 01101111 01101100 01100001

Next, the sender segments the bytes into 2-byte (16-bit) binary numbers:

0100100001101111
0110110001100001

To compute the checksum, the sender adds up the 16-bit binary numbers:

```
  0100100001101111
+ 0110110001100001
  1011010011010000
```

The computer can now send a UDP segment with the encoded "Hola" as the data and 1011010011010000 as the checksum. The entire UDP segment could look like this:

Field	Value
Source port number	00010101 00001001
Destination port number	0001010 100001001
Length	00000000 00000100
Checksum	10110100 11010000
Data	01001000 01101111 01101100 01100001

What if the data got corrupted from "Hola" to "Mola" on the way?

First let's see what the corrupted data would look like in binary.

"Mola" encoded into binary...

M	o	l	a
01001101	01101111	01101100	01100001

...and then segmented into 16-bit numbers:

0100110101101111
0110110001100001

Now let's see what checksum the recipient would compute:

```
  0100110101101111
+0110110001100001
  1011100111010000
```

The recipient can now programmatically compare the checksum they received in the UDP segment with the checksum they just computed:

Received: 1011010011010000

Computed: 1011100111010000

When the recipient discovers that the two checksums are different, it knows that the data was corrupted somehow along the way. Unfortunately, the recipient cannot use the computed checksum to reconstruct the original data, so it will likely just discard the packet entirely.

The actual UDP checksum computation process includes a few more steps than shown here, but this is the general process of how we can use checksums to detect corrupted data.

5.4.2 UDP Use Cases

The Most Common Use Cases of UDP are Video Streaming, DNS, VPN etc.

5.4.3 UDP Pros

- Simple protocol
- Header size is small so datagrams are small
- Uses less bandwidth
- Stateless
- Consumes less memory (no state stored in the server/client)
- Low latency - no handshake, order, retransmission or guaranteed delivery

5.4.4 UDP Cons

- No acknowledgement
- No guarantee delivery
- Connectionless - anyone can send data without prior knowledge

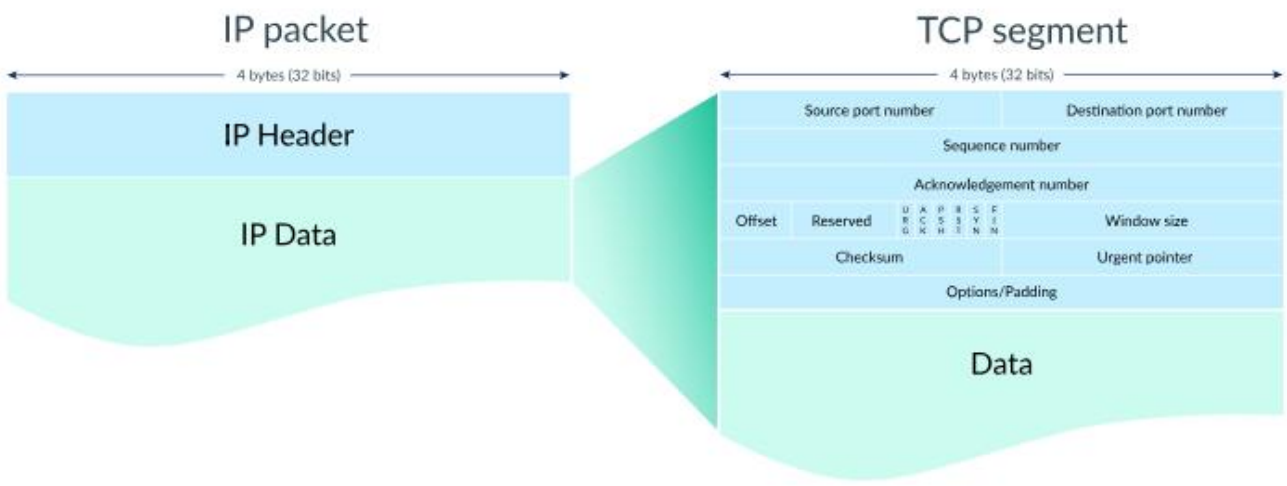
- No flow controls
- No congestion controls
- No ordered packets
- Security - can be easily spoofed

5.5 TCP (Transmission Control Protocol)

The Transmission Control Protocol (TCP) is a transport protocol that is used on top of IP to ensure reliable transmission of packets. TCP includes mechanisms to solve many of the problems that arise from packet-based messaging, such as lost packets, out of order packets, duplicate packets, and corrupted packets. Since TCP is the protocol used most commonly on top of IP, the Internet protocol stack is sometimes referred to as TCP/IP.

5.5.1 Packet format

When sending packets using TCP/IP, the data portion of each IP packet is formatted as a TCP segment.

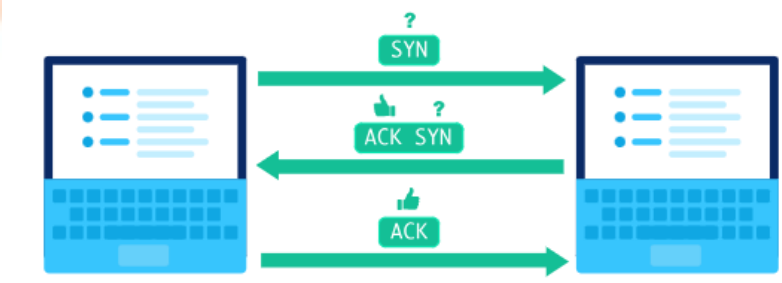


Each TCP segment contains a header and data. The TCP header contains many more fields than the UDP header and can range in size from 20 to 60 bytes, depending on the size of the options field. The TCP header shares some fields with the **UDP** header: Source port number, Destination port number, and Checksum.

5.5.2 Process of transmitting a packet with TCP/IP

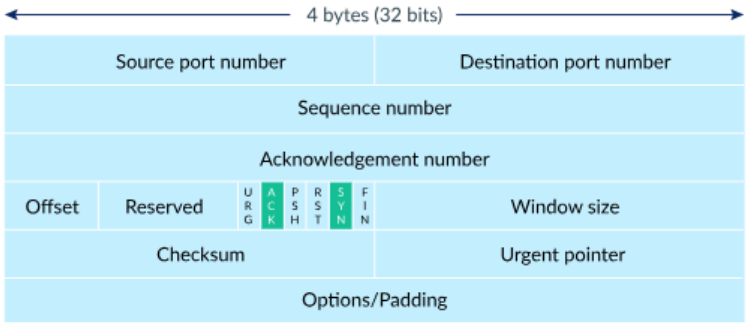
Step 1: Establish connection

When two computers want to send data to each other over TCP, they first need to establish a connection using a three-way handshake.



The first computer sends a packet with the SYN bit set to 1 (SYN = "synchronize?"). The second computer sends back a packet with the ACK bit set to 1 (ACK = "acknowledge!") plus the SYN bit

set to 1. The first computer replies back with an ACK. The SYN and ACK bits are both part of the TCP header:



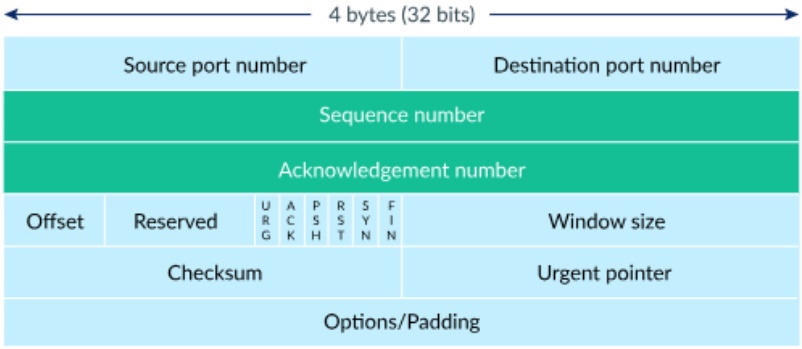
The ACK and SYN bits are highlighted on the fourth row of the header.

In fact, the three packets involved in the three-way handshake do not typically include any data. Once the computers are done with the handshake, they're ready to receive packets containing actual data.

Step 2: Send packets of data

When a packet of data is sent over TCP, the recipient must always acknowledge what they received.

The first computer sends a packet with data and a sequence number. The second computer acknowledges it by setting the ACK bit and increasing the acknowledgement number by the length of the received data. The sequence and acknowledgement numbers are part of the TCP header:



The 32-bit sequence and acknowledgement numbers are highlighted.

Those two numbers help the computers to keep track of which data was successfully received, which data was lost, and which data was accidentally sent twice.

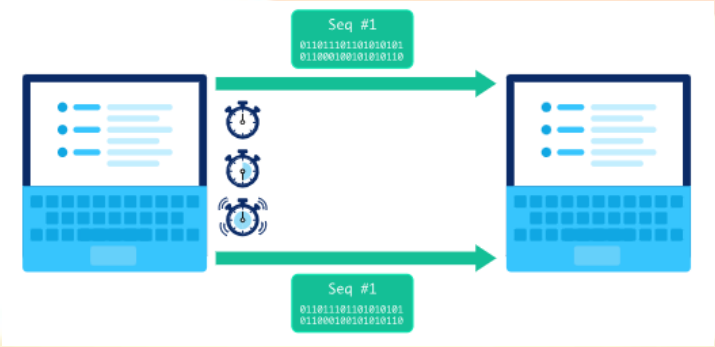
Step 3: Close the connection

Either computer can close the connection when they no longer want to send or receive data.

A computer initiates closing the connection by sending a packet with the FIN bit set to 1 (FIN = finish). The other computer replies with an ACK and another FIN. After one more ACK from the initiating computer, the connection is closed.

5.5.3 Detecting lost packets

TCP connections can detect lost packets using a timeout.



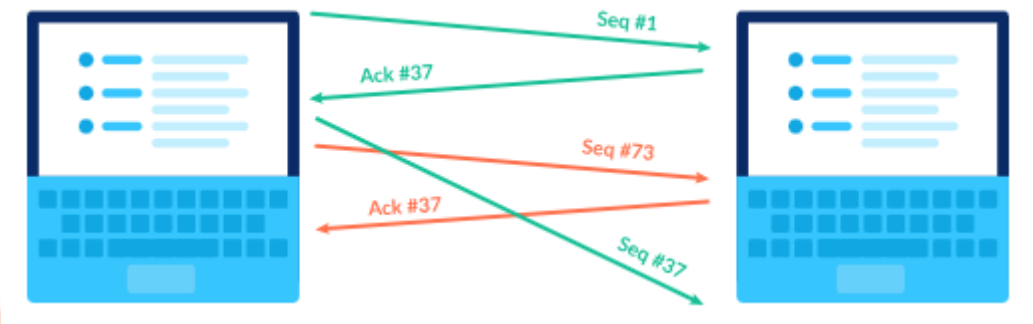
After sending off a packet, the sender starts a timer and puts the packet in a retransmission queue. If the timer runs out and the sender has not yet received an ACK from the recipient, it sends the packet again. The retransmission may lead to the recipient receiving duplicate packets, if a packet was not actually lost but just very slow to arrive or be acknowledged. If so, the recipient can simply discard duplicate packets. It's better to have the data twice than not at all!

5.5.4 Handling out of order packets

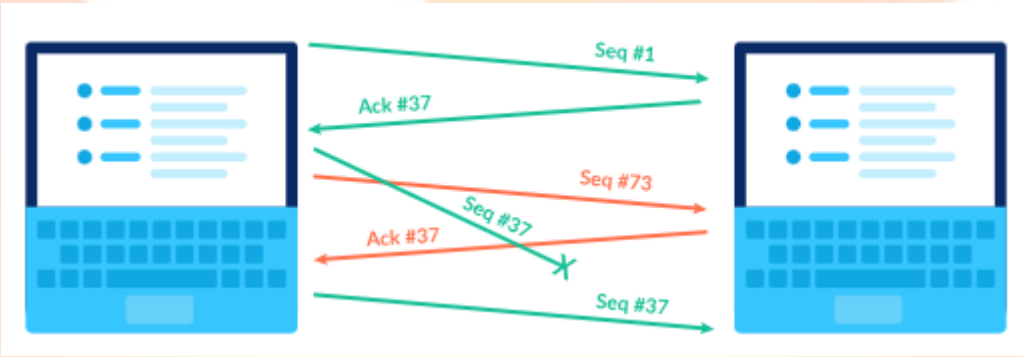
TCP connections can detect out of order packets by using the sequence and acknowledgement numbers.

When the recipient sees a higher sequence number than what they have acknowledged so far, they know that they are missing at least one packet in between. In the situation pictured above, the recipient sees a sequence number of #73 but expected a sequence number of #37. The recipient lets the sender know there's something amiss by sending a packet with an acknowledgement number set to the expected sequence number.

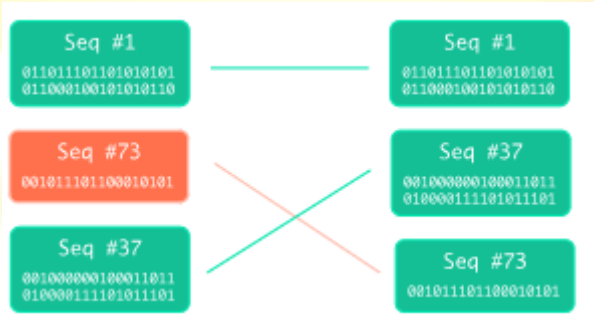
Sometimes the missing packet is simply taking a slower route through the Internet and it arrives soon after.



Other times, the missing packet may actually be a lost packet and the sender must retransmit the packet.



In both situations, the recipient has to deal with out of order packets. Fortunately, the recipient can use the sequence numbers to reassemble the packet data in the correct order.



5.5.5 TCP Pros

- Guarantee delivery
- No one can send data without prior knowledge
- Flow Control and Congestion Control
- Ordered Packets no corruption or app level work
- Secure and can’t be easily spoofed

5.5.6 TCP Cons

- Large header overhead compared to UDP
- More bandwidth
- Stateful - consumes memory on server and client
- Considered high latency for certain workloads (Slow start/ congestion/ acks)
- Does too much at a low level (hence QUIC)
- Single connection to send multiple streams of data (HTTP requests)

- Stream 1 has nothing to do with Stream 2
- Both Stream 1 and Stream 2 packets must arrive
- TCP Meltdown
- Not a good candidate for VPN

5.6 Transport Layer Security (TLS)

Transport Layer Security (TLS) is an Internet Engineering Task Force (IETF) standard protocol that provides authentication, privacy and data integrity between two communicating computer applications. It's the most widely deployed security protocol in use today and is best suited for web browsers and other applications that require data to be securely exchanged over a network. This includes web browsing sessions, file transfers, virtual private network (VPN) connections, remote desktop sessions and voice over IP (VoIP). More recently, TLS is being integrated into modern cellular transport technologies, including 5G, to protect core network functions throughout the radio access network (RAN).

5.6.1 How does Transport Layer Security work?

TLS uses a client-server handshake mechanism to establish an encrypted and secure connection and to ensure the authenticity of the communication. Here's a breakdown of the process:

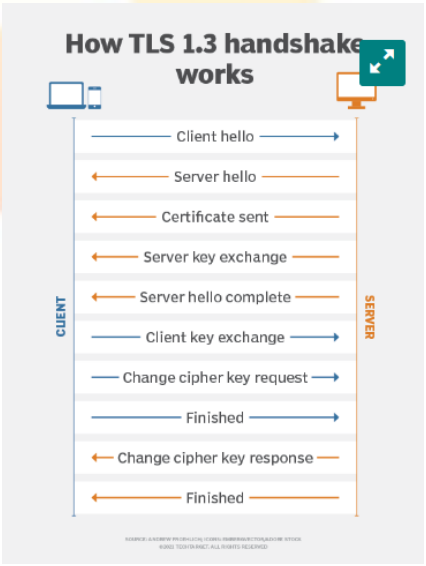
1. Communicating devices exchange encryption capabilities.
2. An authentication process occurs using digital certificates to help prove the server is the entity it claims to be.
3. A session key exchange occurs. During this process, clients and servers must agree on a key to establish the fact that the secure session is indeed between the client and server and not something in the middle attempting to hijack the conversation.

5.6.2 How TLS 1.3 handshake works?

TLS uses a public key exchange process to establish a shared secret between the communicating devices. The two handshake methods are:

- Rivest-Shamir-Adleman (RSA) Handshake
- Diffie-Hellman Handshake

Both methods result in the same goal of establishing a shared secret between communicating devices so the communication can't be hijacked. Once the keys are exchanged, data transmissions between devices on the encrypted session can begin.



5.6.3 History and development of TLS

TLS evolved from Netscape Communications Corp.'s Secure Sockets Layer (SSL) protocol and has largely superseded it, although the terms SSL or SSL/TLS are still sometimes used interchangeably. IEFT officially took over the SSL protocol to standardize it with an open process and released version 3.1 of SSL in 1999 as TLS 1.0. The protocol was renamed TLS to avoid legal issues with Netscape, which developed the SSL protocol as a key part of its original web browser. According to the protocol specification, TLS is composed of two layers: the TLS record protocol and the TLS handshake protocol. The record protocol provides connection security, while the handshake protocol enables the server and client to authenticate each other and to negotiate encryption algorithms and cryptographic keys before any data is exchanged.

The most recent version of TLS, 1.3, was officially finalized by IETF in 2018. The primary benefit over previous versions of the protocol is added encryption mechanisms when establishing a connection handshake between a client and server. While earlier TLS versions offer encryption as well, TLS manages to establish an encrypted session earlier in the handshake process. Additionally, the number of steps required to complete a handshake is reduced, substantially lowering the amount of time it takes to complete a handshake and begin transmitting or receiving data between the client and server.

Another enhancement of TLS 1.3 is that several cryptographic algorithms used to encrypt data were removed, as they were deemed obsolete and weren't recommended for secure transport. Additionally, some security features that were once optional are now required. For example, message-digest algorithm 5 (MD5) cryptographic hashes are no longer supported, perfect forward secrecy (PFS) is required and Rivest Cipher 4 (RC4) negotiation is prohibited. This eliminates the chance that a TLS-encrypted session uses a known insecure encryption algorithm or method in TLS version 1.3.

5.6.4 The benefits of Transport Layer Security

The benefits of TLS are straightforward when discussing using versus not using TLS. As noted above, a TLS-encrypted session provides a secure authentication mechanism, data encryption and data integrity checks. However, when comparing TLS to another secure authentication and encryption protocol suite, such as Internet Protocol Security, TLS offers added benefits and is a reason why IPsec is being replaced with TLS in many enterprise deployment situations. These include benefits such as the following:

- Security is built directly into each application, as opposed to external software or hardware to build IPsec tunnels.
- There is true end-to-end encryption (E2EE) between communicating devices.
- There is granular control over what can be transmitted or received on an encrypted session.
- Since TLS operates within the upper layers of the Open Systems Interconnection (OSI) model, it doesn't have the network address translation (NAT) complications that are inherent with IPsec.
- TLS offers logging and auditing functions that are built directly into the protocol.

5.6.5 The challenges of TLS

There are a few drawbacks when it comes to either not using secure authentication or any encryption or when deciding between TLS and other security protocols, such as IPsec. Here are a few examples:

- Because TLS operates at Layers 4 through 7 of the OSI model, as opposed to Layer 3, which is the case with IPsec, each application and each communication flow between client and server must establish its own TLS session to gain authentication and data encryption benefits.

- The ability to use TLS depends on whether each application supports it.
- Since TLS is implemented on an application-by-application basis to achieve improved granularity and control over encrypted sessions, it comes at the cost of increased management overhead.
- Now that TLS is gaining in popularity, threat actors are more focused on discovering and exploiting potential TLS exploits that can be used to compromise data security and integrity.

5.6.6 Differences between TLS and SSL

As mentioned previously, SSL is the precursor to TLS. Thus, most of the differences between the two are evolutionary in nature, as the protocol adjusts to address vulnerabilities and to improve implementation and integration capabilities.

Key differences between SSL and TLS that make TLS a more secure and efficient protocol are message authentication, key material generation and the supported cipher suites, with TLS supporting newer and more secure algorithms. TLS and SSL are not interoperable, though TLS currently provides some backward compatibility in order to work with legacy systems. Additionally, TLS especially later versions completes the handshake process much faster compared to SSL. Thus, lower communication latency from an end-user perspective is noticeable.

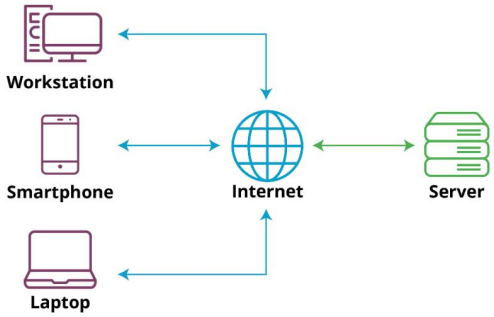
5.6.7 Attacks against TLS/SSL

Implementation flaws have always been a big problem with encryption technologies, and TLS is no exception. Even though TLS/SSL communications are considered highly secure, there have been instances where vulnerabilities were discovered and exploited. But keep in mind that the examples mentioned below were vulnerabilities in TLS version 1.2 and earlier. All known vulnerabilities against prior versions of TLS, such as Browser Exploit Against SSL/TLS (BEAST), Compression Ratio Info-leak Made Easy (CRIME) and protocol downgrade attacks, have been eliminated through TLS version updates. Examples of significant attacks or incidents include the following:

- The infamous Heartbleed bug was the result of a surprisingly small bug vulnerability discovered in a piece of cryptographic logic that relates to OpenSSL's implementation of the TLS heartbeat mechanism, which is designed to keep connections alive even when no data is being transmitted.
- Although TLS isn't vulnerable to the POODLE attack because it specifies that all padding bytes must have the same value and be verified, a variant of the attack has exploited certain implementations of the TLS protocol that don't correctly validate encryption padding byte requirements.
- The BEAST attack was discovered in 2011 and affected version 1.0 of TLS. The attack focused on a vulnerability discovered in the protocol's cipher block chaining (CBC) mechanism. This enabled an attacker to capture and decrypt data being sent and received across the "secure" communications channel.
- An optional data compression feature found within TLS led to the vulnerability known as CRIME. This vulnerability can decrypt communication session cookies using brute-force methods. Once compromised, attackers can insert themselves into the encrypted conversation.
- The Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext (BREACH) vulnerability also uses compression as its exploit target, like CRIME. However, the difference between BREACH and CRIME is the fact that BREACH compromises Hypertext Transfer Protocol (HTTP) compression, as opposed to TLS compression. But, even if TLS compression isn't enabled, BREACH can still compromise the session.

5.7 HTTP/HTTP1.1/HTTP2/HTTP3

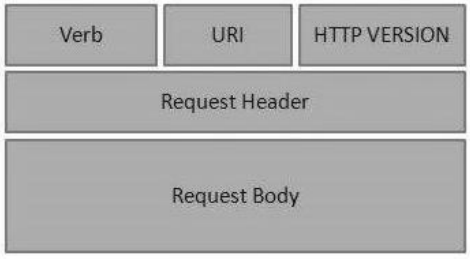
The Hypertext Transfer Protocol (HTTP) is the foundation of the World Wide Web, and is used to load webpages using hypertext links. HTTP is an application layer protocol designed to transfer information between networked devices and runs on top of other layers of the network protocol stack. A typical flow over HTTP involves a client machine making a request to a server, which then sends a response message.



5.7.1 HTTP Request

An HTTP request is the way Internet communications platforms such as web browsers ask for the information, they need to load a website. Each HTTP request made across the Internet carries with it a series of encoded data that carries different types of information. A typical HTTP request contains:

- HTTP version type
- An URL
- An HTTP method
- HTTP request headers
- Optional HTTP body.



HTTP Request

HTTP method

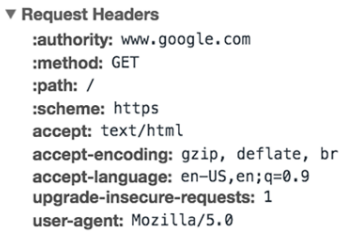
An HTTP method, sometimes referred to as an HTTP verb, indicates the action that the HTTP request expects from the queried server. Following four HTTP methods are commonly used in REST based architecture.

- GET – Provides a read only access to a resource.
- POST – Used to create a new resource.
- DELETE – Used to remove a resource.
- PUT – Used to update an existing resource.
- PATCH – Used to partially-update an existing Resource

HTTP request headers

HTTP headers contain text information stored in key-value pairs, and they are included in every HTTP request (and response, more on that later). These headers communicate core information,

such as what browser the client is using and what data is being requested. Example of HTTP request headers from Google Chrome's network tab:



HTTP request body

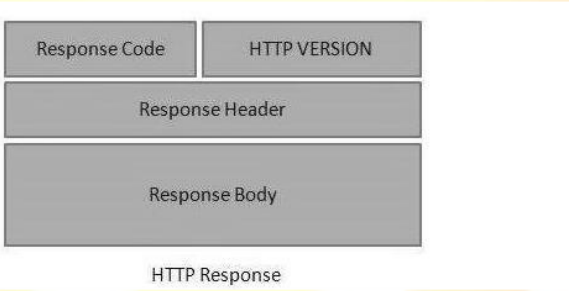
The body of a request is the part that contains the 'body' of information the request is transferring. The body of an HTTP request contains any information being submitted to the web server, such as a username and password, or any other data entered into a form.

5.7.2 HTTP response

An HTTP response is what web clients (often browsers) receive from an Internet server in answer to an HTTP request. These responses communicate valuable information based on what was asked for in the HTTP request.

A typical HTTP response contains:

- HTTP status code
- HTTP response headers
- Optional HTTP body



HTTP status code

HTTP status codes are 3-digit codes most often used to indicate whether an HTTP request has been successfully completed. Status codes are broken into the following 5 blocks:

- 1xx Informational: Communicates transfer protocol-level information.

Status Code	Description
100 Continue	An interim response. Indicates to the client that the initial part of the request has been received and has not yet been rejected by the server. The client SHOULD continue by sending the remainder of the request or, if the request has already been completed, ignore this response. The server MUST send a final response after the request has been completed.
101 Switching Protocol	Sent in response to an Upgrade request header from the client, and indicates the protocol the server is switching to.
102 Processing (WebDAV)	Indicates that the server has received and is processing the request, but no response is available yet.
103 Early Hints	Primarily intended to be used with the `Link` header. It suggests the user agent start preloading the resources while the server prepares a final response.

- 2xx Success: Indicates that the client's request was accepted successfully

Status Code	Description
200 OK	Indicates that the request has succeeded.
201 Created	Indicates that the request has succeeded and a new resource has been created as a result.
202 Accepted	Indicates that the request has been received but not completed yet. It is typically used in log running requests and batch processing.
203 Non-Authoritative Information	Indicates that the returned metainformation in the entity-header is not the definitive set as available from the origin server, but is gathered from a local or a third-party copy. The set presented MAY be a subset or superset of the original version.
204 No Content	The server has fulfilled the request but does not need to return a response body. The server may return the updated meta information.
205 Reset Content	Indicates the client to reset the document which sent this request.
206 Partial Content	It is used when the `Range` header is sent from the client to request only part of a resource.
207 Multi-Status (WebDAV)	An indicator to a client that multiple operations happened, and that the status for each operation can be found in the body of the response.
208 Already Reported (WebDAV)	Allows a client to tell the server that the same resource (with the same binding) was mentioned earlier. It never appears as a true HTTP response code in the status line, and only appears in bodies.
226 IM Used	The server has fulfilled a GET request for the resource, and the response is a representation of the result of one or more instance-manipulations applied to the current instance.

- 3xx Redirection: Indicates that the client must take some additional action in order to complete their request.

Status Code	Description
300 Multiple Choices	The request has more than one possible response. The user-agent or user should choose one of them.
301 Moved Permanently	The URL of the requested resource has been changed permanently. The new URL is given by the `Location` header field in the response. This response is cacheable unless indicated otherwise.
302 Found	The URL of the requested resource has been changed temporarily. The new URL is given by the `Location` field in the response. This response is only cacheable if indicated by a `Cache-Control` or `Expires` header field.
303 See Other	The response can be found under a different URI and SHOULD be retrieved using a GET method on that resource.
304 Not Modified	Indicates the client that the response has not been modified, so the client can continue to use the same cached version of the response.
305 Use Proxy (Deprecated)	Indicates that a requested response must be accessed by a proxy.
306 (Unused)	It is a reserved status code and is not used anymore.
307 Temporary Redirect	Indicates the client to get the requested resource at another URI with same method that was used in the prior request. It is similar to `302 Found` with one exception that the same HTTP method will be used that was used in the prior request.
308 Permanent Redirect (experimental)	Indicates that the resource is now permanently located at another URI, specified by the `Location` header. It is similar to `301 Moved Permanently` with one exception that the same HTTP method will be used that was used in the prior request.

- 4xx Client Error: This category of error status codes points the finger at clients.

Status Code	Description
400 Bad Request	The request could not be understood by the server due to incorrect syntax. The client SHOULD NOT repeat the request without modifications.
401 Unauthorized	Indicates that the request requires user authentication information. The client MAY repeat the request with a suitable Authorization header field
402 Payment Required (Experimental)	Reserved for future use. It is aimed for using in the digital payment systems.
403 Forbidden	Unauthorized request. The client does not have access rights to the content. Unlike 401, the client's identity is known to the server.
404 Not Found	The server can not find the requested resource.
405 Method Not Allowed	The request HTTP method is known by the server but has been disabled and cannot be used for that resource.
406 Not Acceptable	The server doesn't find any content that conforms to the criteria given by the user agent in the `Accept` header sent in the request.
407 Proxy Authentication Required	Indicates that the client must first authenticate itself with the proxy.
408 Request Timeout	Indicates that the server did not receive a complete request from the client within the server's allotted timeout period.
409 Conflict	The request could not be completed due to a conflict with the current state of the resource.
410 Gone	The requested resource is no longer available at the server.
411 Length Required	The server refuses to accept the request without a defined Content- Length. The client MAY repeat the request if it adds a valid `Content-Length` header field.
412 Precondition Failed	The client has indicated preconditions in its headers which the server does not meet.
413 Request Entity Too Large	Request entity is larger than limits defined by server.
414 Request-URI Too Long	The URI requested by the client is longer than the server can interpret.
415 Unsupported Media Type	The media-type in `Content-type` of the request is not supported by the server.
416 Requested Range Not Satisfiable	The range specified by the `Range` header field in the request can't be fulfilled.
417 Expectation Failed	The expectation indicated by the `Expect` request header field can't be met by the server.
418 I'm a teapot (RFC 2324)	It was defined as April's fool joke and is not expected to be implemented by actual HTTP servers. (RFC 2324)
420 Enhance Your Calm (Twitter)	Returned by the Twitter Search and Trends API when the client is being rate limited.
422 Unprocessable Entity (WebDAV)	The server understands the content type and syntax of the request entity, but still server is unable to process the request for some reason.
423 Locked (WebDAV)	The resource that is being accessed is locked.
424 Failed Dependency (WebDAV)	The request failed due to failure of a previous request.

425 Too Early (WebDAV)	Indicates that the server is unwilling to risk processing a request that might be replayed.
426 Upgrade Required	The server refuses to perform the request. The server will process the request after the client upgrades to a different protocol.
428 Precondition Required	The origin server requires the request to be conditional.
429 Too Many Requests	The user has sent too many requests in a given amount of time ("rate limiting").
431 Request Header Fields Too Large	The server is unwilling to process the request because its header fields are too large.
444 No Response (Nginx)	The Nginx server returns no information to the client and closes the connection.
449 Retry With (Microsoft)	The request should be retried after performing the appropriate action.
450 Blocked by Windows Parental Controls (Microsoft)	Windows Parental Controls are turned on and are blocking access to the given webpage.
451 Unavailable For Legal Reasons	The user-agent requested a resource that cannot legally be provided.
499 Client Closed Request (Nginx)	The connection is closed by the client while HTTP server is processing its request, making the server unable to send the HTTP header back.

- 5xx Server Error: The server takes responsibility for these error status codes.

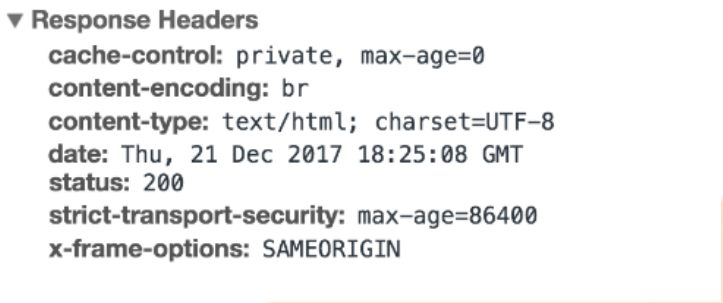
Status Code	Description
500 Internal Server Error	The server encountered an unexpected condition that prevented it from fulfilling the request.
501 Not Implemented	The HTTP method is not supported by the server and cannot be handled.
502 Bad Gateway	The server got an invalid response while working as a gateway to get the response needed to handle the request.
503 Service Unavailable	The server is not ready to handle the request.
504 Gateway Timeout	The server is acting as a gateway and cannot get a response in time for a request.
505 HTTP Version Not Supported (Experimental)	The HTTP version used in the request is not supported by the server.
506 Variant Also Negotiates (Experimental)	Indicates that the server has an internal configuration error: the chosen variant resource is configured to engage in transparent content negotiation itself, and is therefore not a proper endpoint in the negotiation process.
507 Insufficient Storage (WebDAV)	The method could not be performed on the resource because the server is unable to store the representation needed to successfully complete the request.
508 Loop Detected (WebDAV)	The server detected an infinite loop while processing the request.
510 Not Extended	Further extensions to the request are required for the server to fulfill it.
511 Network Authentication Required	Indicates that the client needs to authenticate to gain network access.

The "xx" refers to different numbers between 00 and 99.

Status codes starting with the number '2' indicate a success. For example, after a client requests a webpage, the most commonly seen responses have a status code of '200 OK', indicating that the request was properly completed. If the response starts with a '4' or a '5' that means there was an error and the webpage will not be displayed. A status code that begins with a '4' indicates a client-side error (it is very common to encounter a '404 NOT FOUND' status code when making a typo in a URL). A status code beginning in '5' means something went wrong on the server side. Status codes can also begin with a '1' or a '3', which indicate an informational response and a redirect, respectively.

HTTP response headers

Much like an HTTP request, an HTTP response comes with headers that convey important information such as the language and format of the data being sent in the response body. Example of HTTP response headers from Google Chrome's network tab:

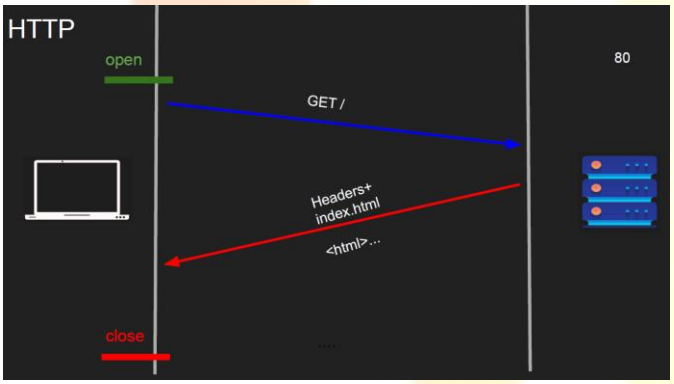


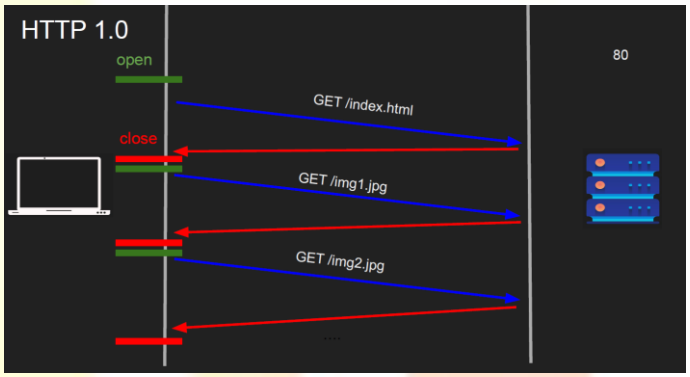
HTTP Response Body

Successful HTTP responses to 'GET' requests generally have a body which contains the requested information. In most web requests, this is HTML data that a web browser will translate into a webpage.

5.7.3 HTTP vs HTTP1.1/HTTP2/HTTP3

In Normal **HTTP** or **HTTP 1.0,** Every request to the same server requires a separate TCP connection for each and every request and this makes the communication slow.



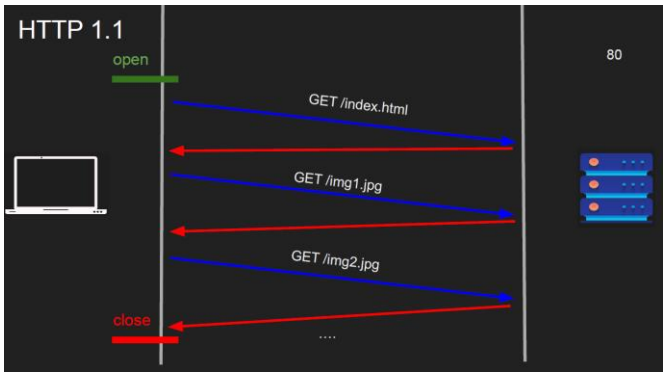


Establishing the connection between the client and the server requires the use of TCP protocol which employs a three-way handshake to establish a communication channel. Although this approach at first improves the reliability between clients and servers, it can lead to performance issues as every HTTP request triggers a TCP three-way handshake which is a time-consuming task. To optimize the performance, there needs to be a solution to reduce the number of TCP connections between clients and servers as fewer connections are created which means less wait time for the clients. This issue was addressed in the introduction of HTTP/1.1.

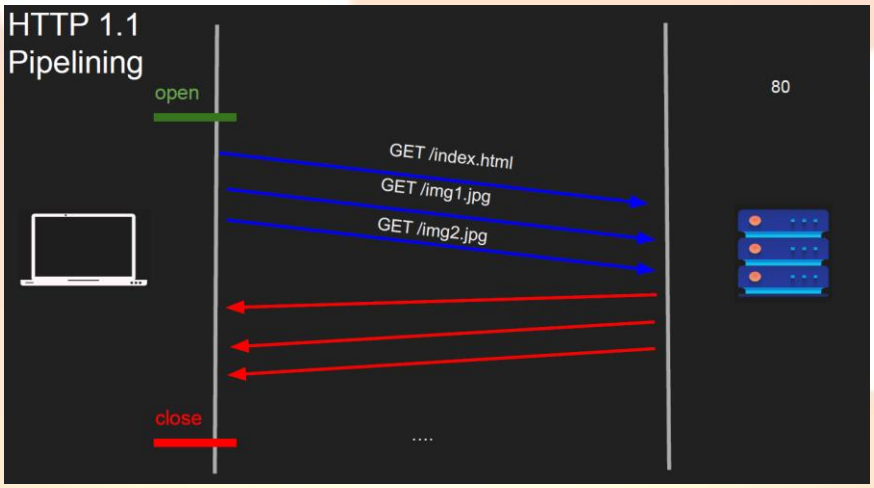
HTTP/1

In **HTTP 1.1**, which was published in 1997 a TCP connection can be left open for reuse (persistent connection), but it doesn't solve the HOL(head-of-line) blocking issue. It has low latency and Low CPU Usage. This version added six new methods: PUT, PATCH, DELETE, CONNECT, TRACE, and OPTIONS. Additionally, a bunch of new caching mechanisms were introduced such as the Cache-Control header, allowing clients and servers to control caching behavior more effectively.

HOL blocking - When the number of allowed parallel requests in the browser is used up, subsequent requests need to wait for the former ones to complete.



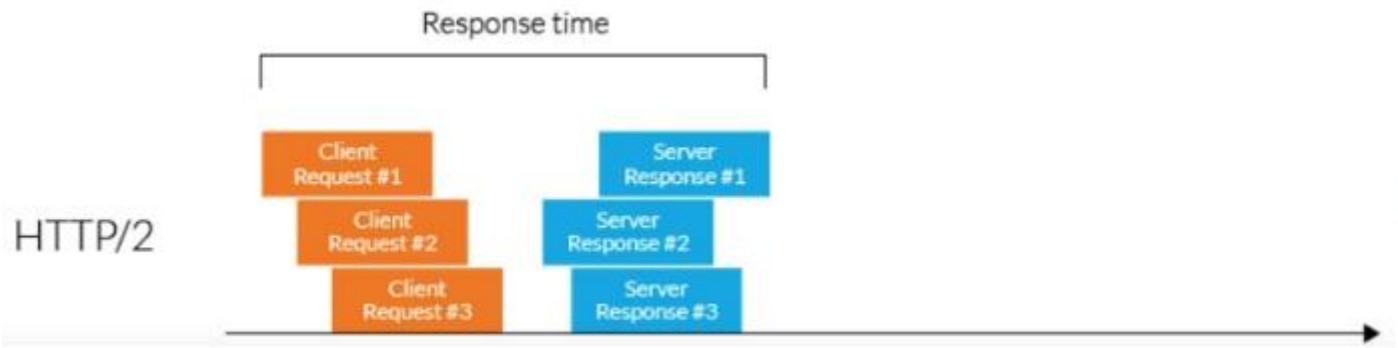
Pipelining is a technique used in HTTP/1.1 to send multiple requests over a single connection without waiting for each response. However, it is limited by head-of-line blocking, which can still cause performance issues.



HTTP/2

In **HTTP 2.0 (HTTP/2)** which was published in 2015, It addresses HOL issue through request multiplexing, which eliminates HOL blocking at the application layer, but HOL still exists at the transport (TCP) layer. HTTP 2.0 introduced the concept of HTTP "streams": an abstraction that allows multiplexing different HTTP exchanges onto the same TCP connection. Each stream doesn't need to be sent in order.

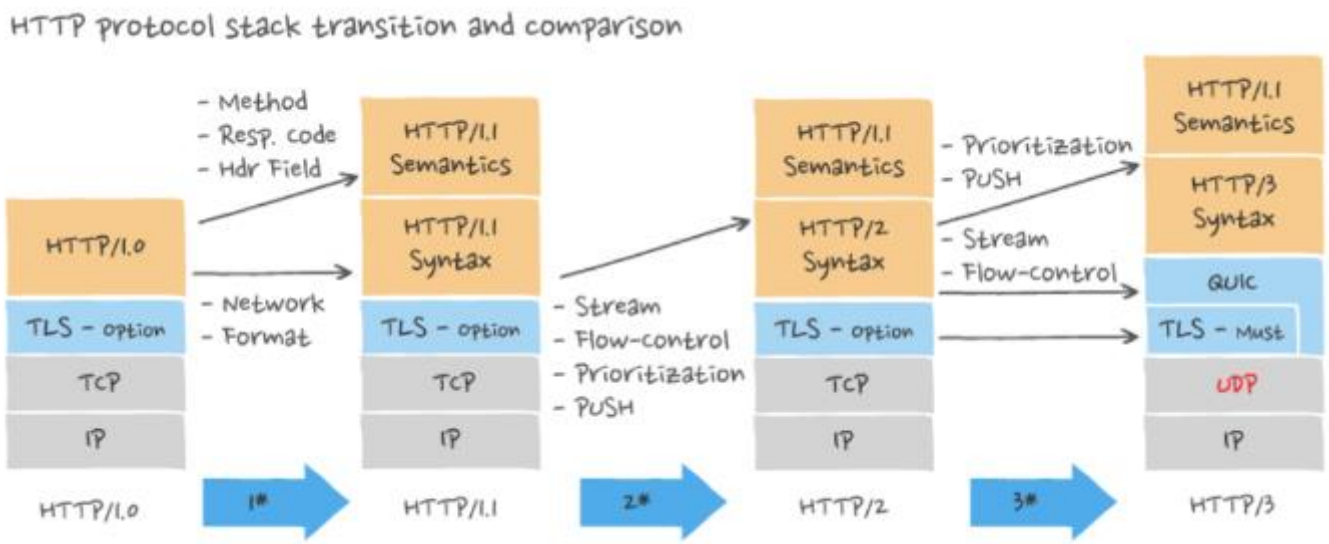
Multiplexing, introduced in HTTP/2, allows multiple requests and responses to be sent concurrently over a single connection, eliminating head-of-line blocking and significantly improving performance.



HTTP/3

The aim of HTTP/3 is to provide fast, reliable, and secure web connections across all forms of devices by straightening out the transport-related issues of HTTP/2. To do this, it uses a different transport layer network protocol called QUIC(Quick UDP Internet Connections), which runs over User Datagram Protocol (UDP) internet protocol instead of the TCP used by all previous versions of HTTP. Unlike TCP's ordered message exchange scheme, UDP allows multidirectional broadcasting of messages, which feature, among other things, helps address head-of-line blocking (HoL) issues at the packet level.

In **HTTP 3.0 (HTTP/3)** which was published in 2020. It is the proposed successor to **HTTP 2.0**. It uses QUIC instead of TCP for the underlying transport protocol, thus removing HOL blocking in the transport layer. QUIC is based on UDP. It introduces streams as first-class citizens at the transport layer. QUIC streams share the same QUIC connection, so no additional handshakes and slow starts are required to create new ones, but QUIC streams are delivered independently such that in most cases packet loss affecting one stream doesn't affect others.



5.8 WebSockets

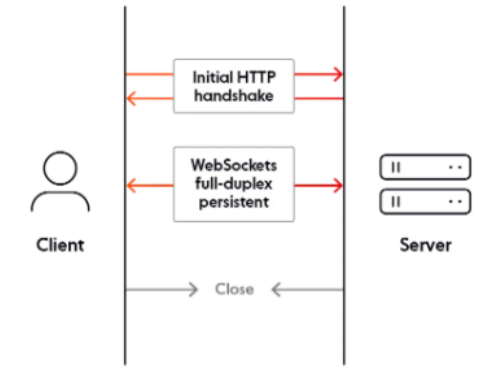
Like HTTP, WebSockets is a communication protocol that enables clients (usually web browsers, hence the name) and servers to communicate with one another. Unlike HTTP with its request-response model, WebSockets are specifically designed to enable real-time bidirectional communication between the server and client. This means the server can push real-time updates (like breaking news) as soon as they become available without waiting for the client to issue a request.

WebSockets is a full-duplex protocol. In simple terms, this means data can flow in both directions over the same connection simultaneously, making WebSockets the go-to choice for applications where the client and server are equally "chatty" and require high throughput. We're talking about things like chat, collaborative editing, and more.

5.8.1 How do WebSockets work?

Even though we are comparing the two, it's important to note that HTTP and WebSockets aren't mutually exclusive. Generally, your application will use HTTP by default then WebSockets for real-time communication code. Interestingly, the way WebSockets work internally is by upgrading the HTTP connection to a WebSocket connection.

When you make a WebSocket connection, WebSockets make a HTTP request to a WebSocket server to ask, "Hey! Do you support WebSockets?" If the server responds "Yes", the HTTP connection is upgraded to a WebSocket connection – this is called an opening handshake.



Sequence for a websocket connection/disconnection

Once the connection is established, both the client and server can transmit and receive data in real-time with minimal latency until either party decides to close the connection.

Note how the URL is prefixed with ws:// for "WebSocket" instead of http:// for "HTTP". Similarly, you would use wss:// instead of https:// when using encryption.

As far as protocols go, WebSockets aren't exactly low level, but they are flexible. The flexibility afforded can be advantageous in specialized situations where you need absolute fine-grain control over your WebSockets code. However, for many developers, the fact that WebSockets are quite barebones is actually a burden because it creates a lot of extra work.

WebSockets don't know how to detect or recover from disconnections. You need to implement something called a heartbeat yourself. Because WebSockets is a protocol completely separate from HTTP, you can't benefit from the value-added stuff HTTP gets from HTTP proxies like compression, for example. You need to decide your own way to implement authentication and error codes compared to HTTP where it's fairly standardized. In many ways, you end up reinventing the wheel as you build out a comprehensive WebSocket messaging layer and that takes time - time you would probably prefer to spend building something unique to your app. Although the WebSocket Web API used in the code example above gives us a handy way to illustrate how WebSockets work, generally, developers sidestep it in favor of open-source libraries that implement all those things like authentication, error handling, compression, etc., for you, as well as patterns like pub/sub or "rooms" to give you a simple way to route messages. These WebSocket libraries tend to focus on the frontend. On the backend, you're still working stateful protocol which makes it tricky to spread work across servers to isolate your app from failures that might lead to congestion (high latency) or even outages. While open-source libraries provide a comprehensive frontend solution, there's usually more work to do on the server if you want to ensure your real time code is robust and reliable with low latency. Increasingly, developers rely on real-time experience platforms like Ably that solve all the annoying data and infrastructure headaches so you can focus on building a fantastic real-time experience for your users.

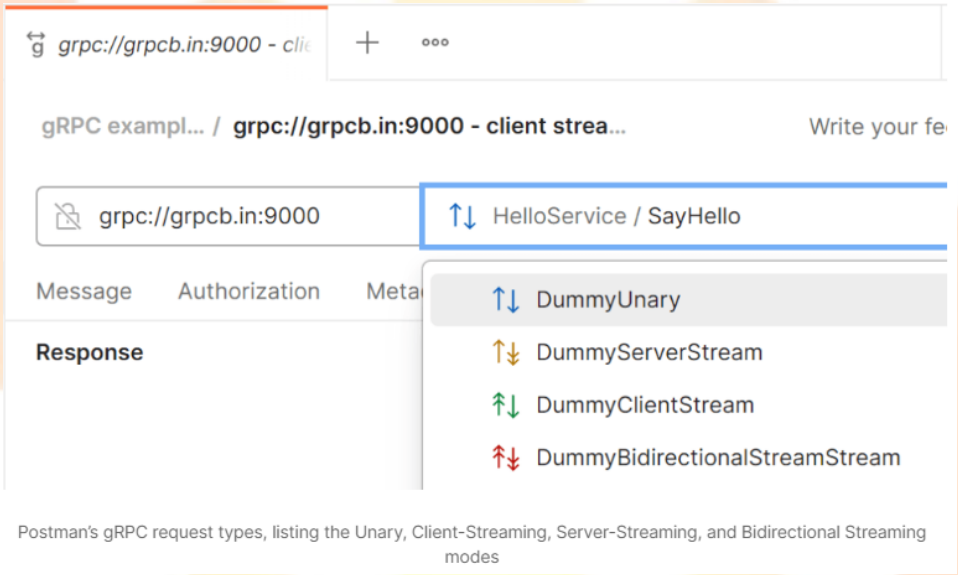
5.9 gRPC

Google developed and then released gRPC in 2016 to achieve higher performance communication between microservices. It is an evolved design of RPC, which was created in the 1970s. Designed with HTTP 2.0, it commonly uses Protocol buffers (Protobufs) for a strictly-binary communication format between the client and server.

The primary concepts of building an RPC API are very similar to building a REST API. Developers define the rules of the interaction between client and server, like the methods to call. Clients submit the call using arguments to invoke the methods, but unlike REST APIs, which use the HTTP methods like GET and POST to define the desired action, RPC APIs define that method in the URL itself, and the query parameters define the instruction to take.

gRPC implements four kinds of service methods for transfer of data, which allow for flexibility in their overall use:

- **Unary**: the client makes a single request, and the server sends a single response, similar to REST over HTTP 1.1.
- **Client-streaming**: the client can send multiple requests to the server, the last of which is an indication that the streamed data is finished, and the server sends back a single response.
- **Server-streaming**: the client sends an initial request alerting the server that it is ready to receive a stream of data, and the server responds with many responses, the last of which indicates that the stream is complete.
- **Bidirectional streaming**: after an initial Unary connection between the client and server, both the client and server can send streams of information.



Postman's gRPC request types, listing the Unary, Client-Streaming, Server-Streaming, and Bidirectional Streaming modes

gRPC allows a client to execute a "remote" (server-based) instruction as though it were part of the local system. Because there is less serialization/deserialization required between systems, gRPC is beneficial to low-power clients such as mobile devices and IoT devices.

Another benefit of gRPC is its "discoverability." As an optional extension, gRPC servers can broadcast a list of requests to clients who request it. This "server reflection" is of tremendous value when working with gRPC APIs. While not a complete replacement for documentation, it does allow easier adoption of an API when developers can get a list of supported instructions. Also, code generation is a feature built into gRPC thanks to its "protoc" compiler.

The main downside for gRPC is adoption, and that adoption is due to the complexity of designing and building a gRPC API. gRPC is generally more difficult to set up and debug since Protocol Buffer messages are binary in nature and not easily human-readable. While RESTful libraries and communication types like JSON are natively supported in browsers, gRPC requires third-party libraries such as gRPC-web, as well as a proxy layer, to perform conversions between HTTP 1.1 and HTTP 2.0.

	HTTP APIs	gRPC APIs
HTTP version	Usually HTTP 1.1	HTTP 2.0
Communication model	Single request/response communication	Single request/response communication, and streamed communication
Browser support	Supported natively	Requires additional library code and proxies
Data transfer	Typically JSON or XML but allows customized choices	Typically uses protocol buffers but allows other types
Data transfer size (relatively speaking)	Medium/large but text can be compressed to binary format	Small due to generally-used binary format
Data typing in payload	JSON is not strongly typed. XML can include "type" data, but increases its size. BSON allows for data typing, but is not widely supported.	Protocol buffers allow strongly-type data transfer
SDK and code generation	Must use third-party tools like Postman to develop code examples for SDK generation	Built-in code generation support
Cross-platform servers	Yes	Yes
Processing complexity	Higher for text-parsing	Lower for well-defined binary structure

5.10 Web RTC

WebRTC (Web Real-time Communication) is an industry effort to enhance the web browsing model. It allows browsers to directly exchange realtime media with other browsers in a peer-to-peer fashion through secure access to input peripherals like webcams and microphones. Traditional web architecture is based on the client-server paradigm, where a client sends an HTTP request to a server and gets a response containing the information requested. In contrast, WebRTC allows the exchange of data among N peers. In this exchange, peers talk to each other without a server in the middle. WebRTC comes built-in with HTML 5, so you don't need a third-party software or plug-in to use it, and you can access it in your browser through the WebRTC API. In this article, you'll learn how and when to use WebRTC, and how it compares to the capabilities of WebSockets.

5.10.1 WebRTC use cases

As a technology, WebRTC has broad applicability. In this section, we'll cover different use cases where WebRTC is a good choice.

Peer-to-peer video, audio, and screen sharing

WebRTC was originally created to facilitate peer-to-peer communication over the internet, especially for video and audio calls. It's now used for more use cases, including text-based chats, file sharing, and screen sharing. WebRTC is used by products like Microsoft Teams, Skype, Slack, and Google Meet. WebRTC has also found relevance in EdTech and healthcare.

File exchange

WebRTC can be used to share files in various formats, even without a video or audio connection. Web Torrent is an example of a file-sharing app built on top of the WebRTC architecture.

Internet of Things

IoT devices are embedded with software and sensors that make it possible to process data and exchange information with other devices on the internet or on a network. WebRTC is helpful when there is a need to send or receive data in real time. For instance, if a drone needs to send video or audio in realtime, WebRTC can make that possible. Surveillance is another IoT example where WebRTC is a strong choice. Think of baby monitors, nanny cams, doorbells, and home cameras. In these scenarios, WebRTC makes it easy to stream video and audio information to smartphones and laptops.

Entertainment and audience engagement

WebRTC is used for entertainment and audience engagement, including augmented reality, virtual reality, and gaming - for example, Google's gaming platform, Stadia, uses WebRTC under the hood.

Realtime language processing

Language processing involves live closed captions, transcriptions, and automatic translations. Through a combination of the HTML5 Speech API and WebRTC's data channels, transcripts can be sent cross-platform in realtime. A good example of this is seen in YouTube and Google Meets, where the closed captions are automatically generated on demand.

5.10.2 Key features of WebRTC

WebRTC consists of several interrelated APIs. We will now look closely at some of the key ones:

- MediaStream
- RTCPeerConnection
- RTCDataChannel

MediaStream

The MediaStream interface is designed to let you access streams of media from local input devices like cameras and microphones. It serves as a way to manage actions on a data stream like recording, sending, resizing, and displaying the stream's content. To use a media stream, the application must request access from the user through the getUserMedia() method. Using this method, you can specify whether you need permission for only video, audio, or both.

To test this on a web page, you need a web server. Opening up an HTML file within the browser won't work because of the security and permission measures that don't allow it to connect to cameras or microphones unless it's being served by an actual server. However, a simple Node.js server can help with this.

RTCPeerConnection

The RTCPeerConnection object is the main entry point to the WebRTC API. It's what allows you to start a connection, connect to peers, and attach media stream information.

Typically, connecting to another browser requires finding where that other browser is located on the web. This is usually in the form of an IP address and a port number, which act as a street address to navigate to your destination. The IP address of your computer or mobile device lets other internet-enabled devices send data directly between each other and is what RTCPeerConnection is built on top of.

RTCDataChannel
The RTCDataChannel API is designed to provide a transport service that allows web browsers to exchange generic data in a bidirectional, peer-to-peer fashion. It uses the Stream Control Transmission Protocol (SCTP) as a way to send data on top of an existing peer connection.

5.11 HTTPS

Hypertext Transfer Protocol Secure (HTTPS) is a protocol that secures communication and data transfer between a user's web browser and a website. HTTPS is the secure version of HTTP. The protocol protects users against eavesdroppers and man-in-the-middle (MitM) attacks. It also protects legitimate domains from domain name system (DNS) spoofing attacks.

HTTPS plays a significant role in securing websites that handle or transfer sensitive data, including data handled by online banking services, email providers, online retailers, healthcare providers and more. Simply put, any website that requires login credentials or involves financial transactions should use HTTPS to ensure the security of users, transactions and data.

5.11.1 HTTP vs. HTTPS

A malicious actor can easily impersonate, modify or monitor an HTTP connection. HTTPS provides protection against these vulnerabilities by encrypting all exchanges between a web browser and web server. As a result, HTTPS ensures that no one can tamper with these transactions, thus securing users' privacy and preventing sensitive information from falling into the wrong hands.

HTTPS is not a separate protocol from HTTP. Rather, it is a variant that uses Transport Layer Security (TLS)/Secure Sockets Layer (SSL) encryption over HTTP to secure communications. When a web server and web browser talk to each other over HTTPS, they engage in what's known as a handshake -- an exchange of TLS/SSL certificates -- to verify the provider's identity and protect the user and their data.

An HTTPS URL begins with https:// instead of http://. Most web browsers show that a website is secure by displaying a closed padlock symbol to the left of the URL in the browser's address bar. In some browsers, users can click on the padlock icon to check if an HTTPS-enabled website's digital certificate includes identifying information about the website owner, such as their name or company name.

5.11.2 How is HTTPS superior to HTTP?

In HTTP, the information shared over a website may be intercepted, or sniffed, by any bad actor snooping on the network. This is especially risky if a user is accessing the website over an unsecured network, such as public Wi-Fi. Since all HTTP communications happen in plaintext, they are highly vulnerable to on-path MitM attacks.

HTTPS ensures that all communications between the user's web browser and a website are completely encrypted. Even if cybercriminals intercept the traffic, what they receive looks like garbled data. This data can be converted to a readable form only with the corresponding decryption tool -- that is, the private key.

5.11.3 Encryption in HTTPS

HTTPS is based on the TLS encryption protocol, which secures communications between two parties. TLS uses asymmetric public key infrastructure for encryption. This means it uses two different keys:

- **The private key**. This is controlled and maintained by the website owner and resides on the web server. It decrypts information that is encrypted by the public key.

- **The public key**. This is available to users who want to securely interact with the server via their web browser. The information encrypted by the public key can only be decrypted by the private key.

5.11.4 How HTTPS works

As noted in the previous section, HTTPS works over SSL/TLS with public key encryption to distribute a shared symmetric key for data encryption and authentication. It uses port 443 by default, whereas HTTP uses port 80. All secure transfers require port 443, although the same port supports HTTP connections as well.

Before a data transfer starts in HTTPS, the browser and the server decide on the connection parameters by performing an SSL/TLS handshake. The handshake is also important to establish a secure connection.

Here's how the entire process works:

- The client browser and the web server exchange "hello" messages.
- Both parties communicate their encryption standards with each other.
- The server shares its certificate with the browser.
- The client verifies the certificate's validity.
- The client uses the public key to generate a pre-master secret key.
- This secret key is encrypted using the public key and shared with the server.
- The client and server compute the symmetric key based on the value of the secret key.
- Both sides confirm that they have computed the secret key.
- Data transmission uses symmetric encryption.

Example of how HTTPS works

Suppose a customer visits a retailer's e-commerce website to purchase an item. When the customer is ready to place an order, they are directed to the product's order page. The URL of this page starts with https://, not http://.

To place the order, the customer is prompted to enter some personal details (e.g., their name and shipping address), as well as financial data (e.g., their credit card number). HTTPS encrypts this data to ensure that it cannot be compromised or stolen by an unauthorized party, such as a hacker or cybercriminal.

The order then reaches the server where it is processed. Once the order is successfully placed, the user receives an acknowledgement from the server, which also travels in encrypted form and displays in their web browser. This acknowledgement is decrypted by the browser's HTTPS sublayer.

5.11.5 HTTPS and the CIA triad

HTTPS guarantees the CIA triad, which is a foundational element in information security:

HTTPS encrypts the website visitor's connection and hides cookies, URLs and other types of sensitive metadata. HTTPS ensures that any data transferred between the visitor and the website cannot be tampered with or modified by a hacker. HTTPS ensures that the user accesses the actual website and not a fake version.

5.11.6 Advantages of HTTPS

HTTPS offers numerous advantages over HTTP connections:

- Data and user protection. HTTPS prevents eavesdropping between web browsers and web servers and establishes secure communications. It thus protects the user's privacy and

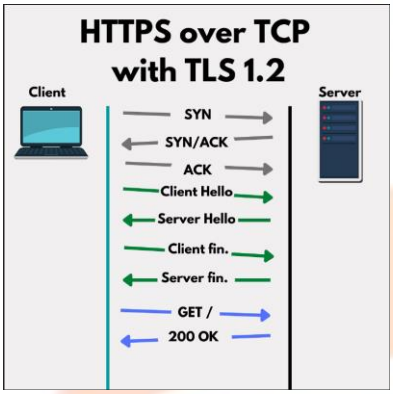
protects sensitive information from hackers. This is critical for transactions involving personal or financial data.

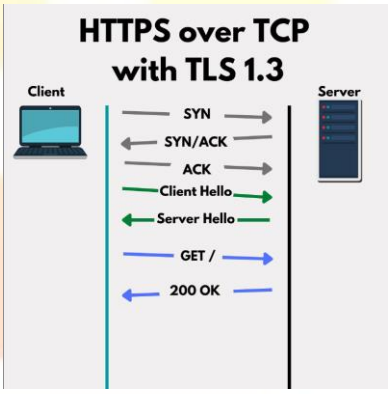
- Improved user experience. When customers know that a website is authentic and protects their data, it instills confidence and trust. In addition, HTTPS increases data transfer speeds by reducing the size of the data.
- Search engine optimization (SEO). HTTPS websites usually rank higher in search engine results pages, which is a significant advantage for organizations looking to boost their digital presence through SEO.

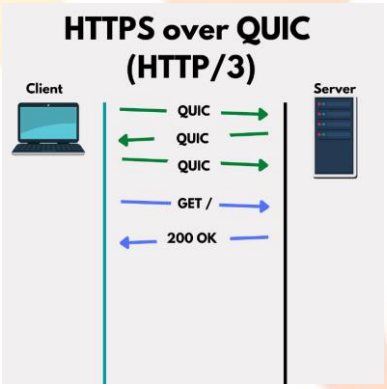
5.11.7 Common mistakes to avoid when adapting HTTPS connection

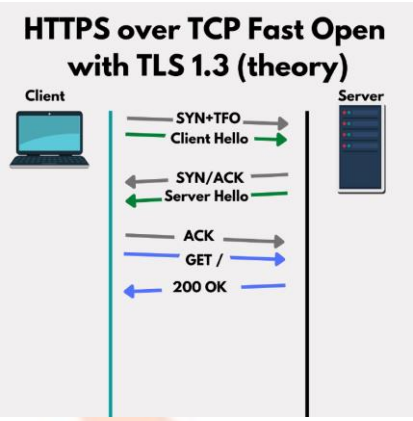
While HTTPS can enhance website security, implementing it improperly can negatively affect a site's security and usability. Common mistakes include the following issues.

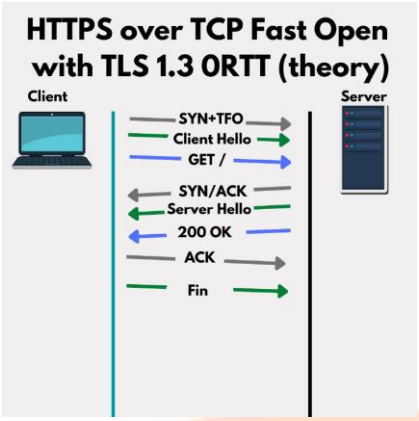
Problem	Solution
Expired certificates	Always ensure that the site certificate is up to date.
Missing certificate for all host names	Get a certificate for all host names that the site serves to avoid certificate name mismatch errors.
Server Name Indication (SNI) support	Ensure that the web server supports SNI and that the audience uses SNI-supported browsers.
Crawling and indexing issues	Ensure that the HTTPS site is not blocked from crawling using robots.txt. Also, enable proper indexing of all pages by search engines.
Content	Ensure that content matches on both HTTP and HTTPS pages.

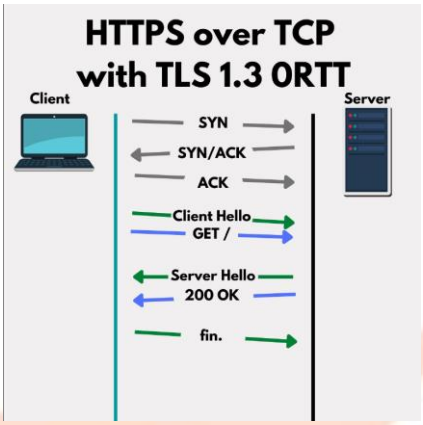


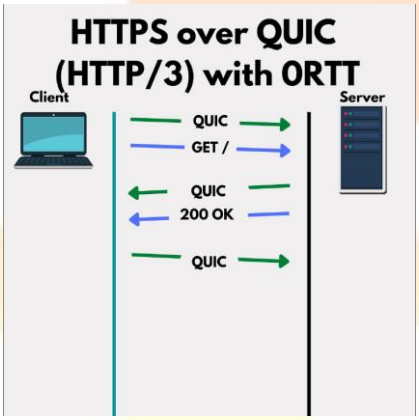












This completes our learning on various ways of Communications and Protocols to Use with the help of which our client and Server can communicated based on the requirement.

6.. Proxy/Reverse Proxy/Load Balancing

When designing complex systems, it's common to use proxy servers to improve performance, security, and reliability. Proxy servers sit between clients and servers and help manage traffic between them.

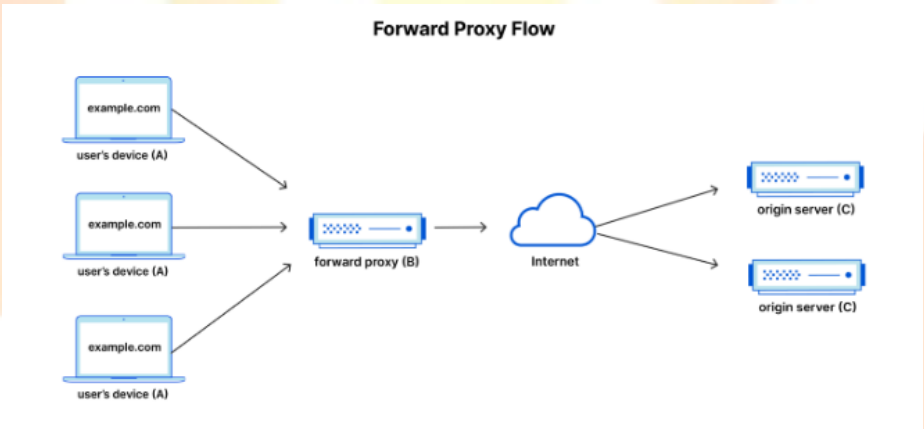
Two types of proxies that are often used are **Forward proxies** and **reverse proxies**. While both are designed to improve the performance and security of a system, they work in different ways and are used in different contexts.

6.1 Forward Proxy or Proxy

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

For example, let's name 3 computers involved in a typical forward proxy communication:

- A: This is a user's home computer
- B: This is a forward proxy server
- C: This is a website's origin server (where the website data is stored)



In a standard Internet communication, computer A would reach out directly to computer C, with the client sending requests to the origin server and the origin server responding to the client. When a forward proxy is in place, A will instead send requests to B, which will then forward the request to C. C will then send a response to B, which will forward the response back to A.

Why would anyone add this extra middleman to their Internet activity? There are a few reasons one might want to use a forward proxy:

- **To avoid state or institutional browsing restrictions** - Some governments, schools, and other organizations use firewalls to give their users access to a limited version of the Internet. A forward proxy can be used to get around these restrictions, as they let the user connect to the proxy rather than directly to the sites they are visiting.
- **To block access to certain content** - Conversely, proxies can also be set up to block a group of users from accessing certain sites. For example, a school network might be configured to connect to the web through a proxy which enables content filtering rules, refusing to forward responses from Facebook and other social media sites.
- **To protect their identity online** - In some cases, regular Internet users simply desire increased anonymity online, but in other cases, Internet users live in places where the government can impose serious consequences to political dissidents. Criticizing the government in a web forum or on social media can lead to fines or imprisonment for these

users. If one of these dissidents uses a forward proxy to connect to a website where they post politically sensitive comments, the IP address used to post the comments will be harder to trace back to the dissident. Only the IP address of the proxy server will be visible.

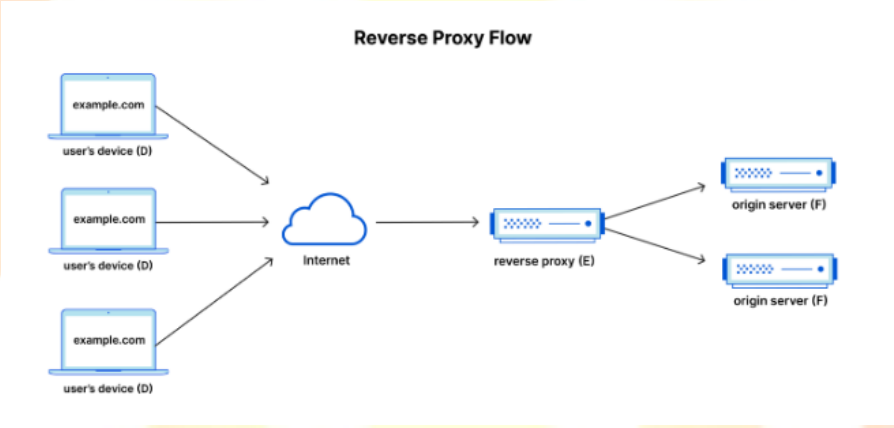
6.2 Reverse Proxy

A reverse proxy is a server that sits in front of one or more web servers, intercepting requests from clients. This is different from a forward proxy, where the proxy sits in front of the clients. With a reverse proxy, when clients send requests to the origin server of a website, those requests are intercepted at the network edge by the reverse proxy server. The reverse proxy server will then send requests to and receive responses from the origin server.

The difference between a forward and reverse proxy is subtle but important. A simplified way to sum it up would be to say that a forward proxy sits in front of a client and ensures that no origin server ever communicates directly with that specific client. On the other hand, a reverse proxy sits in front of an origin server and ensures that no client ever communicates directly with that origin server.

Once again, let's illustrate by naming the computers involved:

- D: Any number of users' home computers
- E: This is a reverse proxy server
- F: One or more origin servers



Typically, all requests from "D" would go directly to "F", and "F" would send responses directly to "D". With a reverse proxy, all requests from "D" will go directly to "E", and "E" will send its requests to and receive responses from "F". "E" will then pass along the appropriate responses to D.

Below we outline some of the benefits of a reverse proxy:

- **Load balancing** - A popular website that gets millions of users every day may not be able to handle all of its incoming site traffic with a single origin server. Instead, the site can be distributed among a pool of different servers, all handling requests for the same site. In this case, a reverse proxy can provide a load balancing solution which will distribute the incoming traffic evenly among the different servers to prevent any single server from becoming overloaded. In the event that a server fails completely, other servers can step up to handle the traffic.
- **Protection from attacks** - With a reverse proxy in place, a web site or service never needs to reveal the IP address of their origin server(s). This makes it much harder for attackers to leverage a targeted attack against them, such as a DDoS attack. Instead, the attackers will only be able to target the reverse proxy, such as Cloudflare's CDN, which will have tighter security and more resources to fend off a cyber-attack.

- **Global server load balancing (GSLB)** - In this form of load balancing, a website can be distributed on several servers around the globe and the reverse proxy will send clients to the server that's geographically closest to them. This decreases the distances that requests and responses need to travel, minimizing load times.
- **Caching** - A reverse proxy can also cache content, resulting in faster performance. For example, if a user in Paris visits a reverse-proxied website with web servers in Los Angeles, the user might actually connect to a local reverse proxy server in Paris, which will then have to communicate with an origin server in L.A. The proxy server can then cache (or temporarily save) the response data. Subsequent Parisian users who browse the site will then get the locally cached version from the Parisian reverse proxy server, resulting in much faster performance.
- **SSL encryption** - Encrypting and decrypting SSL (or TLS) communications for each client can be computationally expensive for an origin server. A reverse proxy can be configured to decrypt all incoming requests and encrypt all outgoing responses, freeing up valuable resources on the origin server.

Some companies build their own reverse proxies, but this requires intensive software and hardware engineering resources, as well as a significant investment in physical hardware. One of the easiest and most cost-effective ways to reap all the benefits of a reverse proxy is by signing up for a **CDN** service. For example, the **Cloudflare CDN** provides all the performance and security features listed above, as well as many others.

6.3 Proxy Server vs API Gateway vs Load Balancer

6.3.1 Load Balancer

A load balancer acts as a mediator between clients and servers, distributing incoming requests across multiple servers in a server farm. It helps achieve scalability, fault tolerance, and improved performance by evenly distributing the workload. Load balancers use various algorithms to determine which server should handle each request, considering factors like server health, available resources, and session persistence.

Use Cases:

- **Web Application Scaling**: Load balancers are widely used in scenarios where there is a need to scale web applications by distributing incoming traffic across multiple servers. This ensures efficient resource utilization and improves application performance.
- **High Availability**: Load balancers help achieve high availability by detecting server failures and redirecting traffic to healthy servers. This ensures that the application remains accessible even if individual servers go down.
- **Handling Peak Loads**: Load balancers are valuable during periods of high traffic or peak loads. They distribute the incoming requests across multiple servers, preventing any single server from being overwhelmed and ensuring that all requests are processed efficiently.

Benefits:

- **Improved Performance**: Load balancers optimize resource utilization by evenly distributing traffic, leading to improved response times and better overall performance of web applications.
- **Scalability**: Load balancers allow for seamless horizontal scaling by adding more servers to the server farm. This helps accommodate increased traffic and ensures that the application can handle growing user demands.
- **Enhanced Reliability**: With load balancers, the risk of application downtime due to server failures is minimized. Load balancers detect and redirect traffic away from failed servers, ensuring continuous availability and a reliable user experience.

6.3.2 Reverse Proxy

A reverse proxy, also known as an application-level gateway, sits between clients and servers and handles requests on behalf of the servers. It acts as an intermediary for client requests, forwarding them to the appropriate server and returning the server's response to the client. Reverse proxies provide benefits such as caching, SSL termination, and improved security by shielding servers from direct exposure to the internet.

Use Cases:

- **Web Application Security**: Reverse proxies act as a protective shield for backend servers by intercepting and filtering incoming requests. They can provide security measures such as filtering malicious traffic, protecting against DDoS attacks, and implementing access controls.
- **Caching and Content Delivery**: Reverse proxies can cache static and dynamic content, reducing the load on backend servers and improving response times for subsequent requests. This is particularly useful for websites with high traffic and frequent content updates.
- **SSL Termination**: Reverse proxies can handle SSL encryption and decryption, offloading this task from backend servers. This simplifies the server configuration and reduces the computational load on the servers, improving performance.

Benefits:

- **Enhanced Security**: Reverse proxies provide an additional layer of security by shielding backend servers from direct exposure to the internet. They can filter and block malicious requests, preventing attacks and unauthorized access.
- **Improved Performance**: By caching and delivering content, reverse proxies reduce the load on backend servers, resulting in faster response times and improved overall performance for users.
- **Flexibility in Application Deployment**: Reverse proxies enable organizations to deploy multiple web applications on the same server or server farm. They can route requests based on URL patterns or domain names, allowing for efficient hosting of multiple applications on a single infrastructure.

6.3.3 API Gateway

An API gateway is an entry point for clients to access backend services, or APIs. It acts as a single point of entry, abstracting the underlying architecture and providing a unified interface for clients. API gateways offer various functionalities like request routing, protocol translation, authentication, and rate limiting. They enable organizations to manage and secure their APIs effectively.

Use Cases:

- **API Management**: API gateways are commonly used to manage and expose APIs to clients. They provide a centralized entry point for accessing APIs, allowing organizations to control and monitor API traffic effectively.
- **Protocol Translation**: API gateways can act as a bridge between different protocols, translating requests and responses between clients and backend services. This enables interoperability between systems that use different communication protocols.
- **Security and Authentication**: API gateways provide security features such as authentication, authorization, and access control. They can enforce API usage policies, validate client credentials, and protect against unauthorized access to backend services.

Benefits:

- **Simplified API Integration**: API gateways offer a unified interface for clients to access multiple backend services or APIs. They abstract the underlying architecture, making it easier for clients to integrate with and consume different services.
- **Scalability and Performance**: API gateways can handle high volumes of API traffic by distributing requests across multiple backend servers. They provide features like request throttling and caching to improve performance and handle spikes in demand.
- **Analytics and Monitoring**: API gateways often include analytics and monitoring capabilities, allowing organizations to track API usage, monitor performance metrics, and gain insights into

6.3.4 Load Balancer vs Reverse Proxy

While both load balancers and reverse proxies handle network traffic distribution, they operate at different layers of the network stack. Load balancers primarily focus on distributing traffic across servers based on server load, availability, and other factors. On the other hand, reverse proxies handle requests on behalf of servers, providing additional functionalities such as caching, SSL termination, and security enhancements.

6.3.5 Load Balancer vs API Gateway

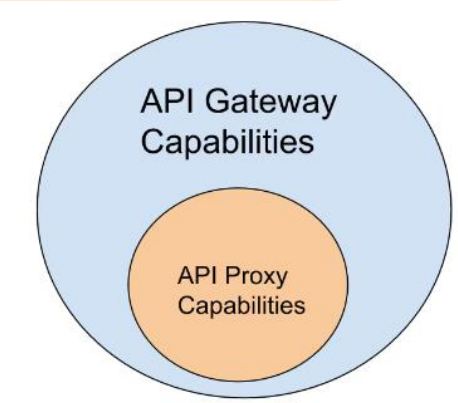
Load balancers and API gateways serve different purposes in the context of web application architectures. Load balancers distribute traffic across servers, ensuring high availability and improved performance. API gateways, on the other hand, provide a centralized entry point for APIs, allowing organizations to manage and secure their API ecosystem effectively.

6.3.6 Reverse Proxy vs API Gateway

While both reverse proxies and API gateways act as intermediaries between clients and servers, they have distinct roles. Reverse proxies primarily focus on handling client requests and forwarding them to the appropriate server, while providing additional functionalities like caching and SSL termination. API gateways, on the other hand, focus on managing APIs, including request routing, authentication, and rate limiting.

6.3.7 Forward proxy vs API gateway

Both the Forward proxy and the API gateway sit in front of your client machines, acting like gatekeepers, but with a big difference in capabilities. In most cases, an API gateway can do everything the Forward proxy does and more. In terms of capability and as shown in the diagram below, the API proxy only has a subset of the capabilities that the API gateway possesses.



6.3.8 Considerations for Choosing the Right Solution

When choosing between a load balancer, reverse proxy, or API gateway, several factors should be considered:

- **Use Case**: Evaluate your specific requirements and understand the primary purpose you need to fulfil. Determine if you need to distribute traffic, enhance security, manage APIs, or a combination of these functionalities.
- **Scalability**: Consider the scalability requirements of your application. If you anticipate significant growth or fluctuations in traffic, choose a solution that can handle the expected load and seamlessly scale as needed.
- **Performance**: Assess the performance impact of each solution. Consider factors such as response times, throughput, and the ability to handle peak loads efficiently.
- **Security**: Determine the level of security needed for your application. If you require advanced security features like DDoS protection, web application firewall (WAF), or SSL offloading, a reverse proxy or API gateway may be more suitable.
- **Integration**: Evaluate how well each solution integrates with your existing infrastructure, backend services, and development frameworks. Compatibility and ease of integration play a vital role in successful implementation.
- **Management and Monitoring**: Consider the management and monitoring capabilities offered by each solution. Look for features such as centralized configuration, analytics, logging, and alerting that align with your operational requirements.
- **Cost**: Assess the cost implications of each solution, including initial setup, licensing, ongoing maintenance, and potential scalability costs. Determine the solution that offers the best value for your budget.

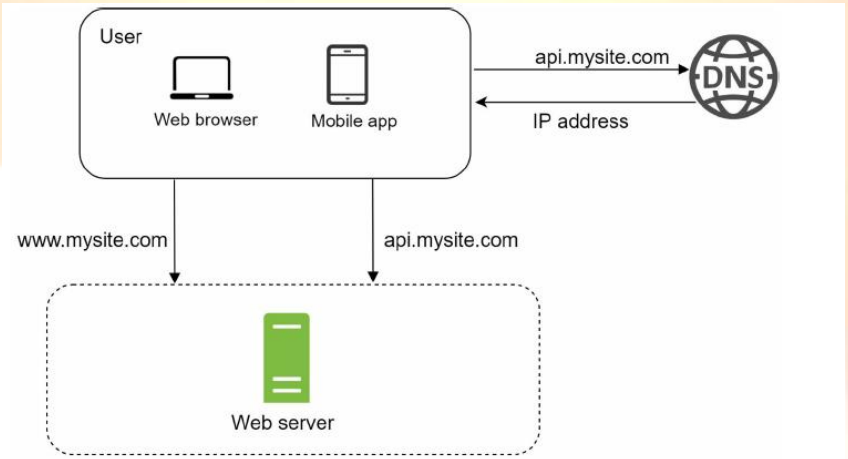
7..

8.. Scaling Any Application from Zero to Millions of Users

Scaling of any applications from Zero to Millions of Users involves many steps, which begins with the Creation of Complete Application in One Server.

8.1 Setup Single Server

Single Server means everything i.e., Cache, Database Server, Web Application, Static Files etc lies in the one server. Ideally, it is 2 Tier Client Server Architecture.



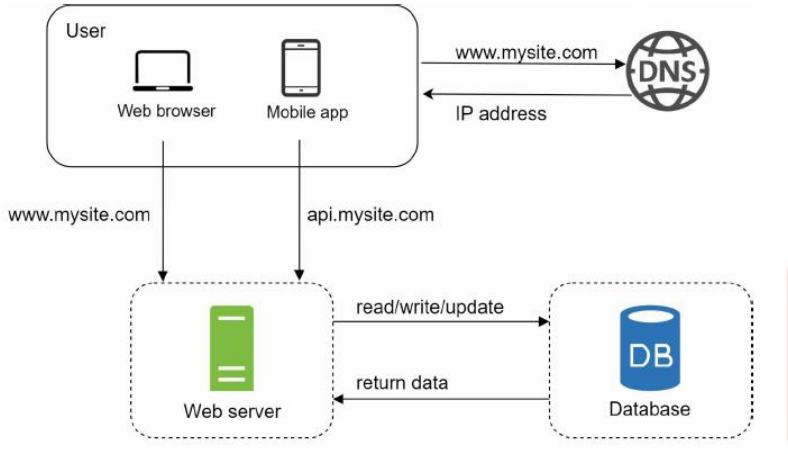
1. Here, Client (Web Browser or Mobile Application) will send a Request to **Domain Name System (DNS)** First with the URL which User wants to Access. Suppose, User here wants to access **api.mysite.com**. Usually, the **DNS** is a paid service provided by 3rd parties and not hosted by our servers. And, DNS will look up for **Internet Protocol (IP) address** of Requested URL and that IP Address is returned to the browser or mobile app. In the example, IP address 15.125.23.214 is returned.
2. Once the IP address is obtained, Hypertext Transfer Protocol (HTTP) requests are sent directly to the web server.
3. The web server returns HTML pages or JSON response for rendering.

Note: Traffic to Any server comes from either Web or Mobile Application

8.2 Separate Database Server from Application Server

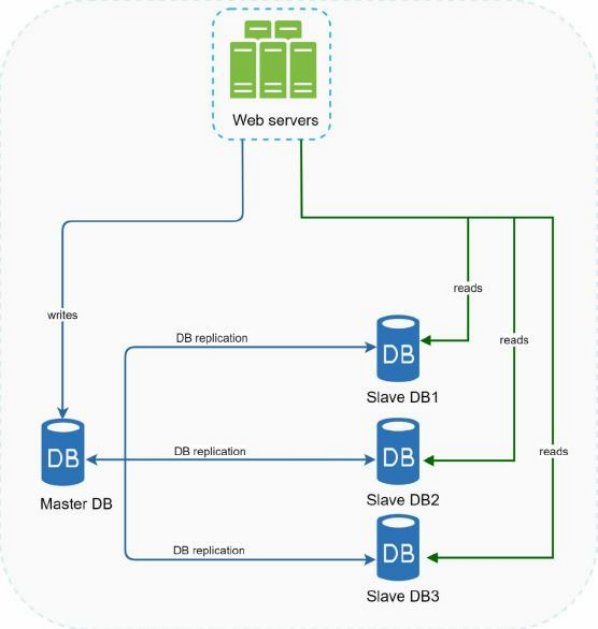
With the growth in the number of users accessing our application, one server is not enough, and we need multiple servers: one for handling web/mobile traffic, the other for handling the Database

traffic. So, we will be Separating web/mobile traffic (Application Server or Web Tier) and database (Database Server or Data Tier) so that each of these servers can be scaled independently. Ideally, we are converting 2-Tier Client Server Architecture to 3-Tier Client Server Architecture.



8.3 Scale up Database Server Independently (Database Replication)

Database Replication can be used in many DBMS usually with a Master-Slave Database Concepts between **Master** (Original Database) and Various Replicated Databases called **Slaves**. All Write Operations i.e., Commands modifying Data in the Database like Insert, Update or Delete must be Processed by Master Database while All Read operations must be processed by Slave Databases as it has all copies of Data from Master. Since, In Any applications Read Operations Gets Triggered more as compared to Write Operation, so that's the reason number of Slave Databases should be More than That of Master Database.



Points to Remember:

- If there is only One Slave DB and it goes offline, then Read operations will be directed to Master DB Temporarily. As soon as issue is found, old slave DB will be replaced by New One. While in case of multiple slave DB's if one goes offline, then Read operations will be directed to another healthy slave DB and a new DB Server will replace the old one.

- If Master DB goes offline because of any reason, then a Slave DB will be promoted to Master DB and all DB write operations will be executed on this new Master DB. A new Slave DB will replace the Old Slave for Data Replication Immediately. In production this is quite

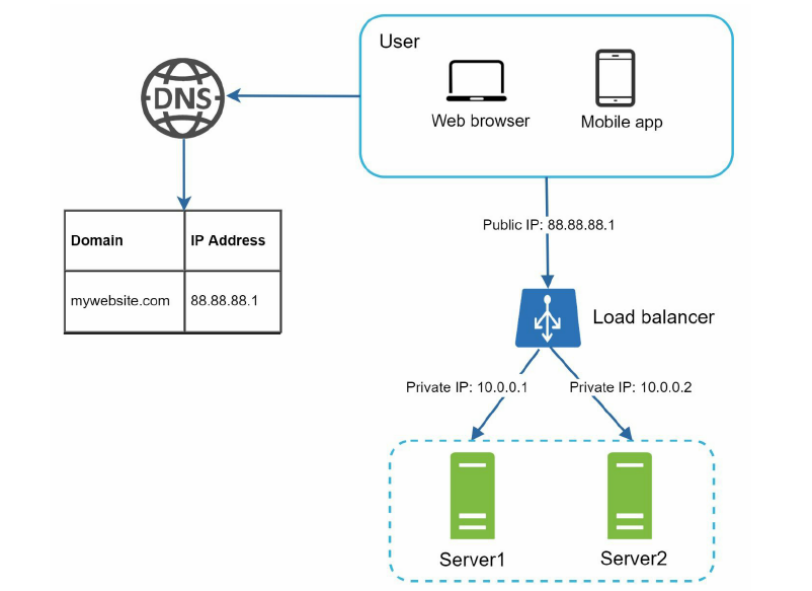
complicated as the Slave DB which is getting Promoted to Master DB might not be up to date. Missing data needs to be updated by running data recovery steps manually.

Advantages of database replication

- **Better performance**: In the master-slave model, all writes and updates happen in master nodes; whereas, read operations are distributed across slave nodes. This model improves performance because it allows more queries to be processed in parallel.
- **Reliability**: If one of your database servers is destroyed by a natural disaster, such as a typhoon or an earthquake, data is still preserved. You do not need to worry about data loss because data is replicated across multiple locations.
- **High availability**: By replicating data across different locations, your website remains in operation even if a database is offline as you can access data stored in another database server.

8.4 Scale up Application Server Independently (Load Balancer)

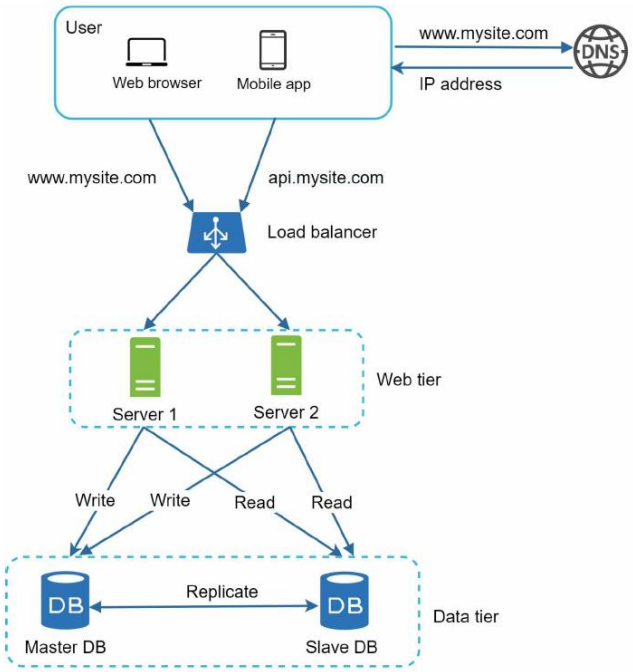
Consider a situation where users are connected to the one application server directly. In this case user will not be able to access the Web Site if the server is low or offline or if many users beyond the limit tries to access the application server simultaneously i.e., a very high traffic is observed. In this case User will face slow response or fail to connect to the server. A load balancer is the best way to address these problems, it will distribute incoming traffic among N web Servers.



Here, client connects with Load Balancer through public IP address returned from DNS for any URL. Now, all the application servers are not reachable for the client as Load Balancer uses private IP's to establish a communication with application servers. This makes application servers more Secure.

Now, if any of the application servers are down then traffic will be routed to other healthy servers and this prevents web sites from going offline. We can add or remove application servers to/from the server pool if traffic increases/decreases rapidly.

The system design after adding the load balancer and database replication



8.5 Reducing Response/Load Time

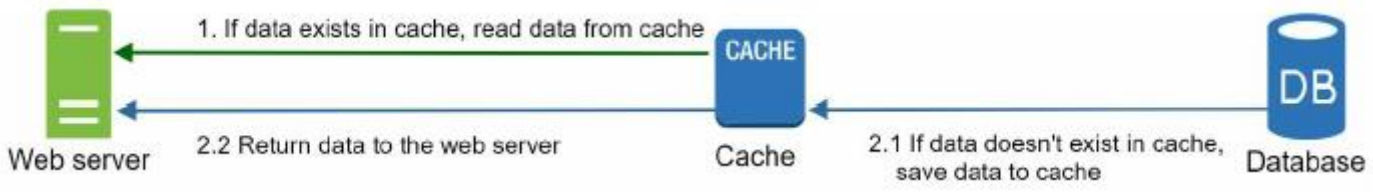
We have scaled up application servers and DB Server to handle the situations which can make the application offline. Now, we have to improve response and load times. For this, we will introduce:

1. Cache or Cache Tier
2. CDN (Content Delivery Network)

8.5.1 Cache Tier

A **Cache** is a temporary Storage area that stores the result of any expensive responses or frequently accessed data in memory so that subsequent requests are served more quickly. Every time a new web page loads, one or more database calls are executed to fetch data. The application performance is greatly affected by calling the database repeatedly. The cache can mitigate this problem.

The **Cache Tier** is a temporary data store layer, much faster than the database. The benefits of having a separate cache tier include better system performance, ability to reduce database workloads, and the ability to scale the cache tier independently.



After receiving a request, a web server first checks if the cache has the available response. If it has, it sends data back to the client. If not, it queries the database, stores the response in cache, and sends it back to the client. This caching strategy is called a read-through cache. Other caching strategies are available depending on the data type, size, and access patterns. Interacting with cache servers is simple because most cache servers provide APIs for common programming languages. The following code snippet shows typical Memcached APIs:

```
SECONDS = 1
cache.set('myKey', 'hi there', 3600 * SECONDS)
cache.get('myKey')
```

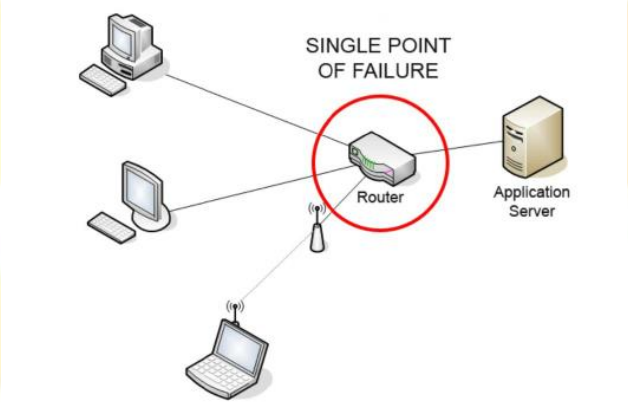
Considerations for Using Cache

- **Decide when to use cache**: Consider using cache when data is read frequently but modified infrequently. Since cached data is stored in volatile memory, a cache server is not ideal for persisting data. For instance, if a cache server restarts, all the data in memory is lost. Thus, important data should be saved in persistent data stores.

- **Expiration Policy**: Expiration Policy should be there. i.e., Once cached data is expired, it is removed from the cache. When there is no expiration policy, cached data will be stored in the memory permanently. It is advisable not to make the expiration date too short as this will cause the system to reload data from the database too frequently. Meanwhile, it is advisable not to make the expiration date too long as the data can become stale.

- **Consistency**: This involves keeping the data store and the cache in sync. Inconsistency can happen because data-modifying operations on the data store and cache are not in a single transaction. When scaling across multiple regions, maintaining consistency between the data store and cache is challenging.

- **Mitigating Failures**: A single cache server represents a potential single point of failure (SPOF), defined in Wikipedia as follows: "A single point of failure (SPOF) is a part of a system that, if it fails, will stop the entire system from working". As a result, multiple cache servers across different data centres are recommended to avoid SPOF.

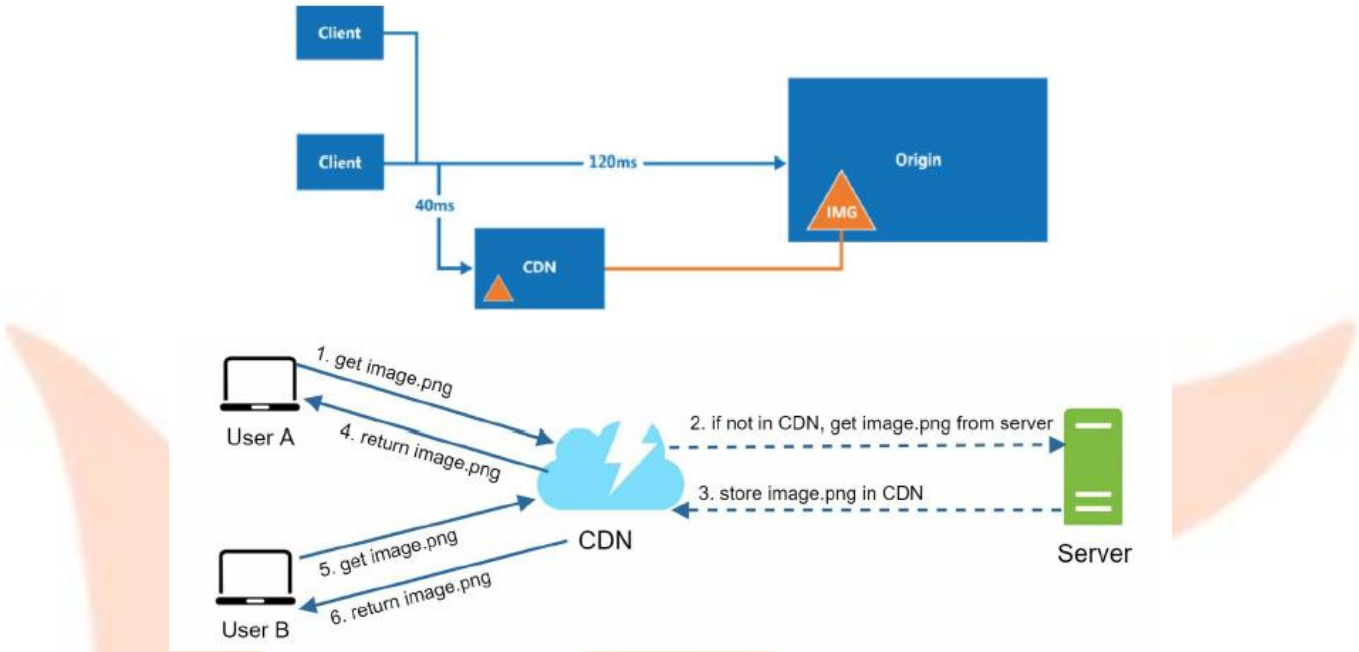


- **Eviction Policy**: Once the cache is full, any requests to add items to the cache might cause existing items to be removed. This is called cache eviction. Least-recently-used (LRU) is the most popular cache eviction policy. Other eviction policies, such as the Least Frequently Used (LFU) or First in First Out (FIFO), can be adopted to satisfy different use cases.

8.5.2 Content Delivery Network (CDN)

A CDN is a network of geographically dispersed servers used to deliver static content. CDN servers cache static content like images, videos, CSS, JavaScript files, etc.

Here is how CDN works at the high-level: when a user visits a website, a CDN server closest to the user will deliver static content. Intuitively, the further users are from CDN servers, the slower the website loads. For example, if CDN servers are in San Francisco, users in Los Angeles will get content faster than users in Europe.



1. User A tries to get image.png by using an image URL. The URL's domain is provided by the CDN provider. The following two image URLs are samples used to demonstrate what image URLs look like on Amazon and Akamai CDNs:

- https://mysite.cloudfront.net/logo.jpg
- https://mysite.akamai.com/image-manager/img/logo.jpg

2. If the CDN server does not have image.png in the cache, the CDN server requests the file from the origin, which can be a web server or online storage like Amazon S3.

3. The origin returns image.png to the CDN server, which includes optional HTTP header Time-to-Live (TTL) which describes how long the image is cached.

4. The CDN caches the image and returns it to User A. The image remains cached in the CDN until the TTL expires.

5. User B sends a request to get the same image.

6. The image is returned from the cache as long as the TTL has not expired.

Considerations while setting up CDN

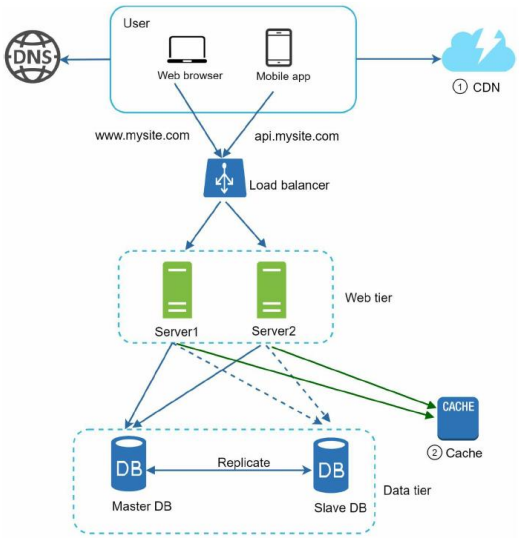
- **Cost**: CDNs are run by third-party providers, and you are charged for data transfers in and out of the CDN. Caching infrequently used assets provides no significant benefits so you should consider moving them out of the CDN.

- **Setting an appropriate cache expiry:** For time-sensitive content, setting a cache expiry time is important. The cache expiry time should neither be too long nor too short. If it is too long, the content might no longer be fresh. If it is too short, it can cause repeat reloading of content from origin servers to the CDN.

- **CDN fallback**: You should consider how your website/application copes with CDN failure. If there is a temporary CDN outage, clients should be able to detect the problem and request resources from the origin.

- **Invalidating files:** You can remove a file from the CDN before it expires by performing one of the following operations: Invalidate the CDN object using APIs provided by CDN vendors or Use object versioning to serve a different version of the object. To version an object, you can add a parameter to the URL, such as a version number. For example, version number 2 is added to the query string: image.png?v=2.

The System design after the CDN and cache are added.

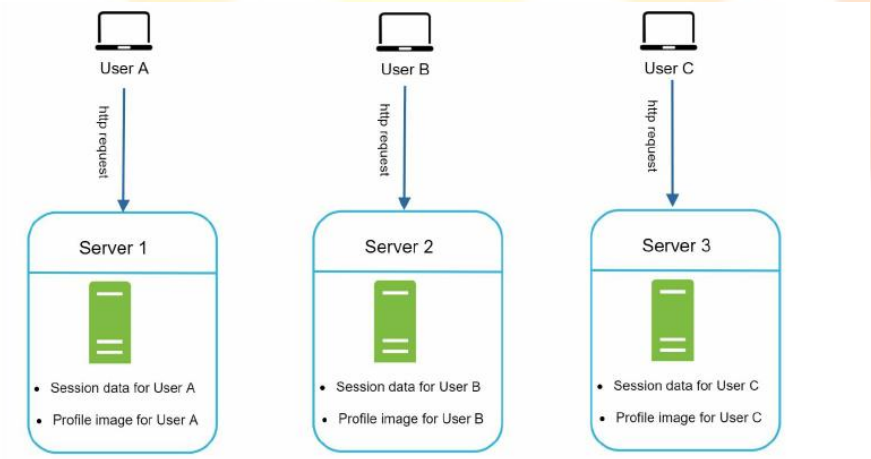


8.6 Scaling Web Tier Horizontally

Now it is time to consider scaling the web tier horizontally. For this, we need to move state (for instance user session data) out of the web tier. A good practice is to store session data in the persistent storage such as relational database or NoSQL. Each web server in the cluster can access state data from databases. This is called **Stateless Web Tier**. We have already discussed about Stateful and Stateless Architecture already to achieve this. Just let's re-brief it here.

8.6.1 Stateful architecture

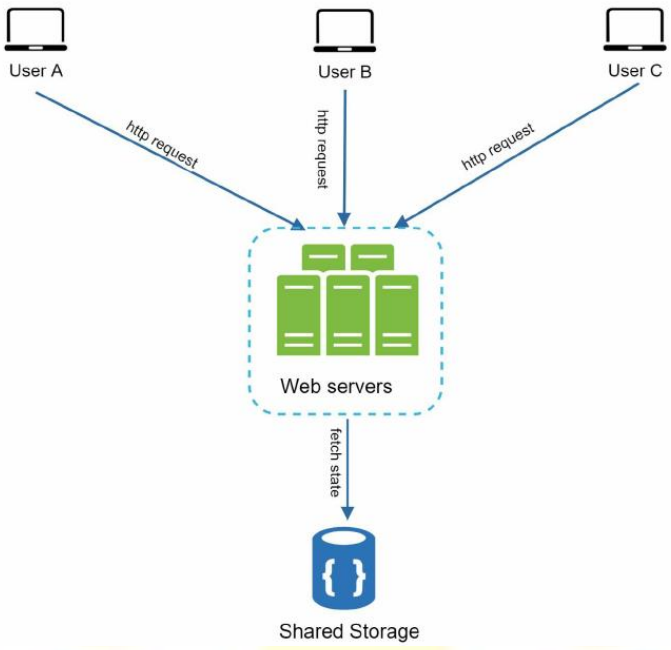
A stateful server and stateless server has some key differences. A stateful server remembers client data (state) from one request to the next. A stateless server keeps no state information.



Here, user A's session data and profile image are stored in Server 1. To authenticate User A, HTTP requests must be routed to Server 1. If a request is sent to other servers like Server 2, authentication would fail because Server 2 does not contain User A's session data. Similarly, all HTTP requests from User B must be routed to Server 2; all requests from User C must be sent to Server 3. The issue is that every request from the same client must be routed to the same server. This can be done with sticky sessions in most load balancers; however, this adds the overhead.

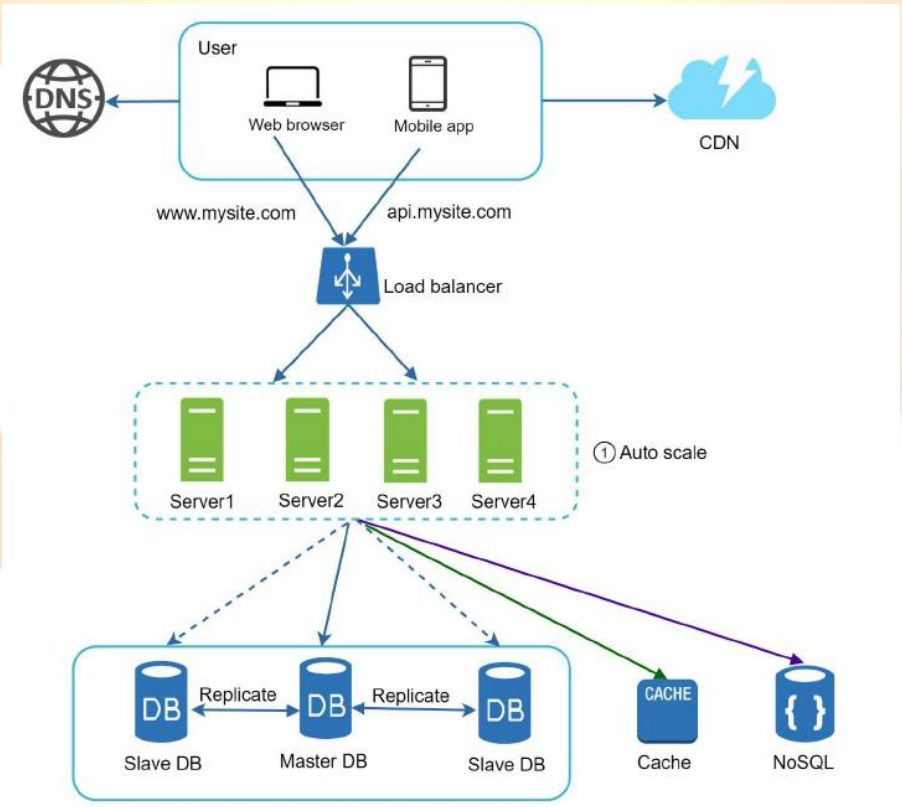
Adding or removing servers is much more difficult with this approach. It is also challenging to handle server failures. To overcome this, we should always go for **Stateless Architecture**.

8.6.2 Stateless Architecture



In this stateless architecture, HTTP requests from users can be sent to any web servers, which fetch state data from a shared data store. State data is stored in a shared data store and kept out of web servers. A stateless system is simpler, more robust, and scalable.

The updated system design with a stateless web tier:

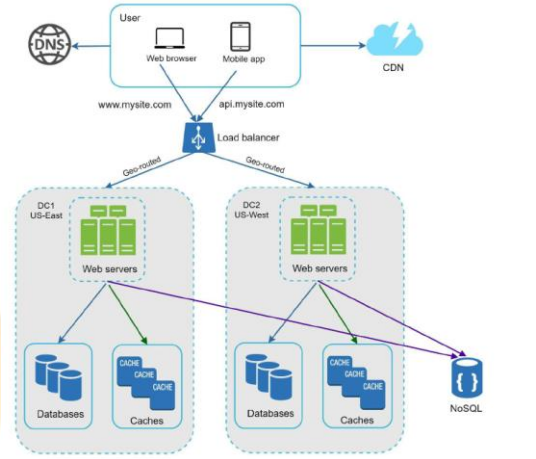


We move the session data out of the web tier and store them in the persistent data store. The shared data store could be a relational database, Memcached/Redis, NoSQL, etc. The NoSQL data store is chosen as it is easy to scale. Autoscaling means adding or removing web servers

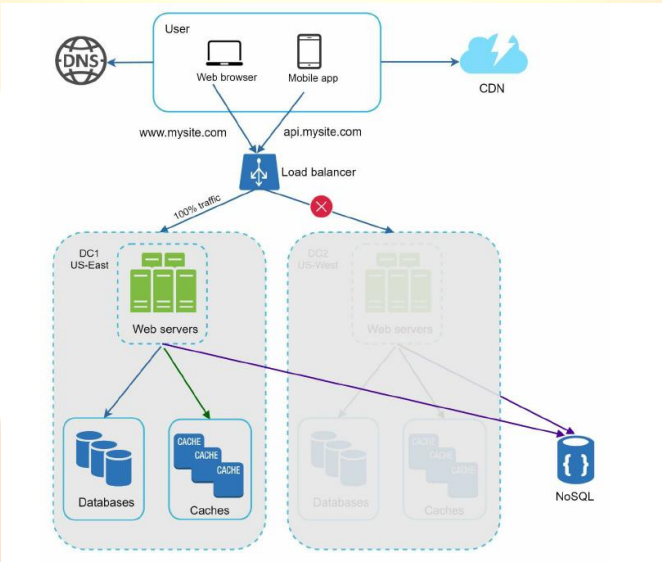
automatically based on the traffic load. After the state data is removed out of web servers, auto-scaling of the web tier is easily achieved by adding or removing servers based on traffic load.

8.7 Data Centres

Your website grows rapidly and attracts a significant number of users internationally. To improve availability and provide a better user experience across wider geographical areas, supporting multiple Data Centres is crucial. In normal operation, users are geo-routed, to the closest data centre, with a split traffic of x% in US-East and (100 – x)% in US-West. **geoDNS** is a DNS service that allows domain names to be resolved to IP addresses based on the location of a user.



In the event of any significant Data Center outage, we direct all traffic to a healthy Data Center. In above system, data centre 2 (US-West) is offline, and 100% of the traffic is routed to data centre 1 (US-East).



Various Challenges in Setting Up Data Centres

- **Traffic redirection**: Effective tools are needed to direct traffic to the correct data centre. GeoDNS can be used to direct traffic to the nearest data centre depending on where a user is located.

- **Data synchronization**: Users from different regions could use different local databases or caches. In failover cases, traffic might be routed to a data centre where data is unavailable. A common strategy is to replicate data across multiple data centres. A previous study shows how Netflix implements asynchronous multi-data centre replication.

- **Test and deployment**: With multi-data centre setup, it is important to test your website/application at different locations. Automated deployment tools are vital to keep services consistent through all the data centres.

8.8 Decouple Different Components of the System

To further scale our system, we need to decouple different components of the system so they can be scaled independently. **Messaging queue** is a key strategy employed by many real-world distributed systems to solve this problem.

8.8.1 Message Queue

A message queue is a durable component, stored in memory, that supports asynchronous communication. It serves as a buffer and distributes asynchronous requests. The basic architecture of a message queue is simple. Input services, called **producers/publishers**, create messages, and publish them to a message queue. Other services or servers, called **consumers/subscribers**, connect to the queue, and perform actions defined by the messages.

Decoupling makes the message queue a preferred architecture for building a scalable and reliable application. With the message queue, the producer can post a message to the queue when the consumer is unavailable to process it. The consumer can read messages from the queue even when the producer is unavailable.

Consider the following use case: your application supports photo customization, including cropping, sharpening, blurring, etc. Those customization tasks take time to complete. Now, Web Servers will publish photo processing jobs to the message queue. Photo processing workers pick up jobs from the message queue and asynchronously perform photo customization tasks. The producer and the consumer can be scaled independently. When the size of the queue becomes large, more workers are added to reduce the processing time. However, if the queue is empty most of the time, the number of workers can be reduced.

8.8.2 Logging, Metrics, Automation

When working with a small website that runs on a few servers, logging, metrics, and automation support are good practices but not a necessity. However, now that your site has grown to serve a large business, investing in those tools is essential.

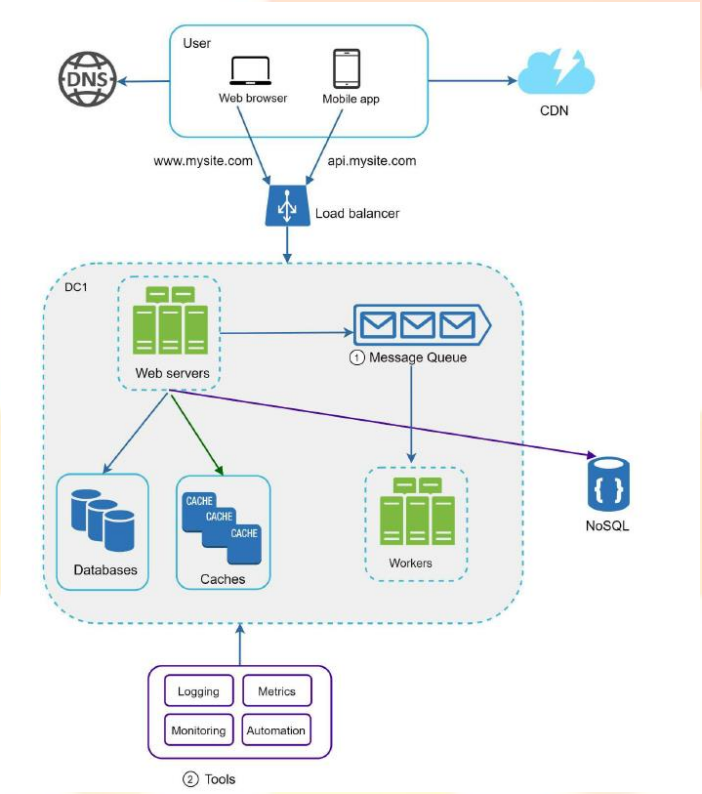
Logging: Monitoring error logs is important because it helps to identify errors and problems in the system. You can monitor error logs at per server level or use tools to aggregate them to a centralized service for easy search and viewing.

Metrics: Collecting different types of metrics help us to gain business insights and understand the health status of the system. Some of the following metrics are useful:

- Host level metrics: CPU, Memory, disk I/O, etc.
- Aggregated level metrics: for example, the performance of the entire database tier, cache tier, etc.
- Key business metrics: daily active users, retention, revenue, etc.

Automation: When a system gets big and complex, we need to build or leverage automation tools to improve productivity. Continuous integration is a good practice, in which each code check-in is verified through automation, allowing teams to detect problems early. Besides, automating your build, test, deploy process, etc. could improve developer productivity significantly.

Updated System Design with Message Queue and Logging, Metrics and Automation.



8.9 Database Scaling

There are two broad approaches for database scaling:

- Vertical Scaling
- Horizontal Scaling.

8.9.1 Vertical scaling

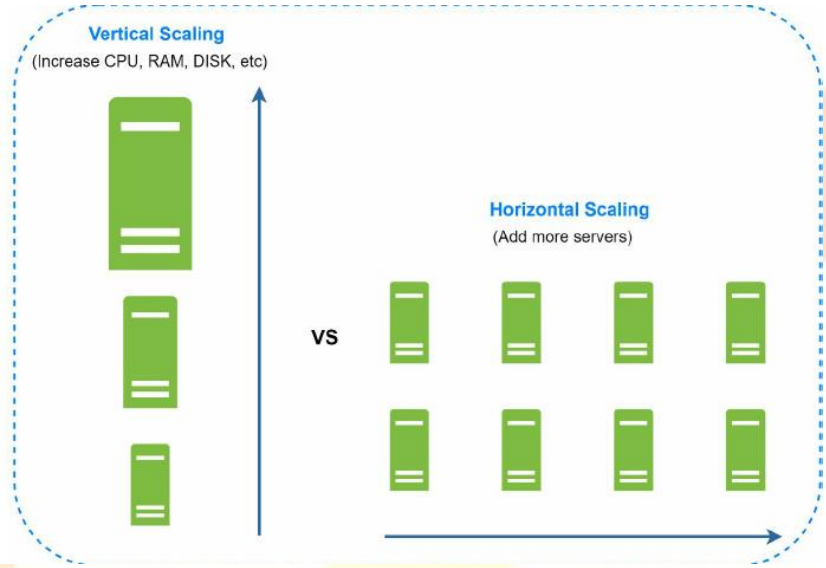
Vertical Scaling, also known as scaling up, is the scaling by adding more power (CPU, RAM, DISK, etc.) to an existing machine. There are some powerful database servers. According to Amazon Relational Database Service (RDS) [12], you can get a database server with 24 TB of RAM. This kind of powerful database server could store and handle lots of data. For example, stackoverflow.com in 2013 had over 10 million monthly unique visitors, but it only had 1 master database [13]. However, vertical scaling comes with some serious drawbacks:

- You can add more CPU, RAM, etc. to your database server, but there are hardware limits. If you have a large user base, a single server is not enough.
- Greater risk of single point of failures.

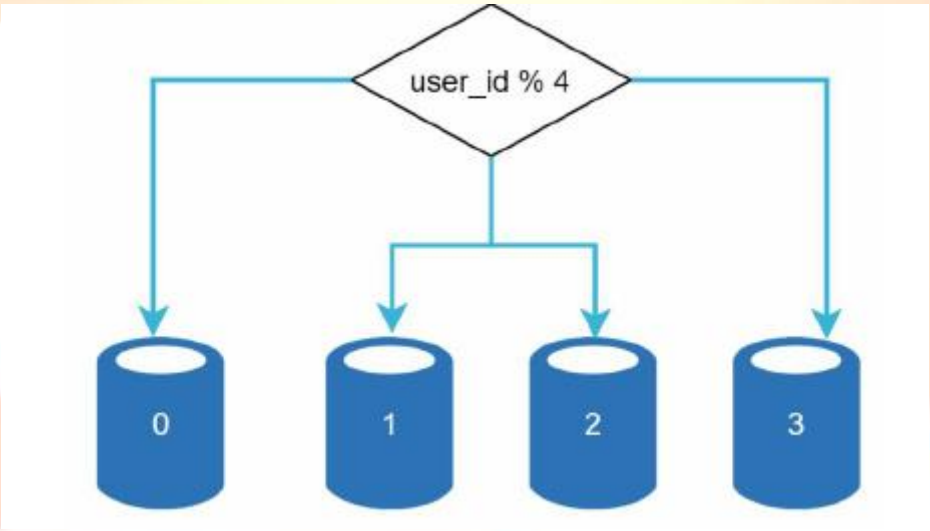
- The overall cost of vertical scaling is high. Powerful servers are much more expensive.

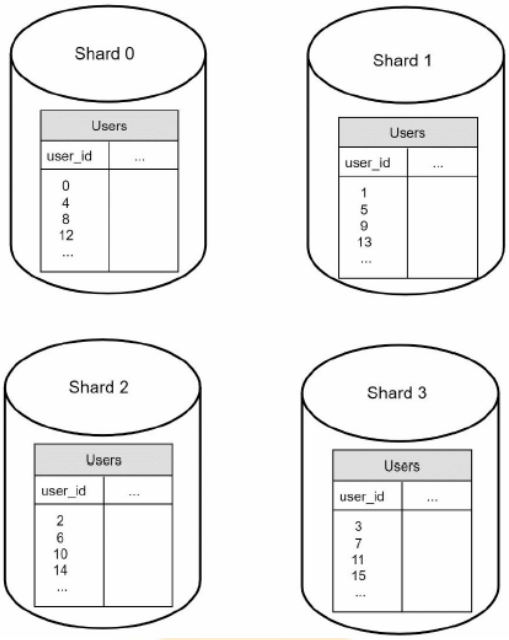
8.9.2 Horizontal Scaling

Horizontal Scaling, also known as Sharding, is the practice of adding more servers. **Sharding** separates large databases into smaller, more easily managed parts called **shards**. Each shard shares the same schema, though the actual data on each shard is unique to the **shard**.



An Example of sharded databases. User data is allocated to a database server based on user IDs. Anytime you access data, a hash function is used to find the corresponding shard. In our example, user_id % 4 is used as the hash function. If the result equals to 0, shard 0 is used to store and fetch data. If the result equals to 1, shard 1 is used. The same logic applies to other shards.





The most important factor to consider when implementing a sharding strategy is the choice of the sharding key. Sharding key (known as a partition key) consists of one or more columns that determine how data is distributed. As shown in Figure 1-22, "user_id" is the sharding key. A sharding key allows you to retrieve and modify data efficiently by routing database queries to the correct database. When choosing a sharding key, one of the most important criteria is to choose a key that can evenly distributed data. Sharding is a great technique to scale the database but it is far from a perfect solution. It introduces complexities and new challenges to the system.