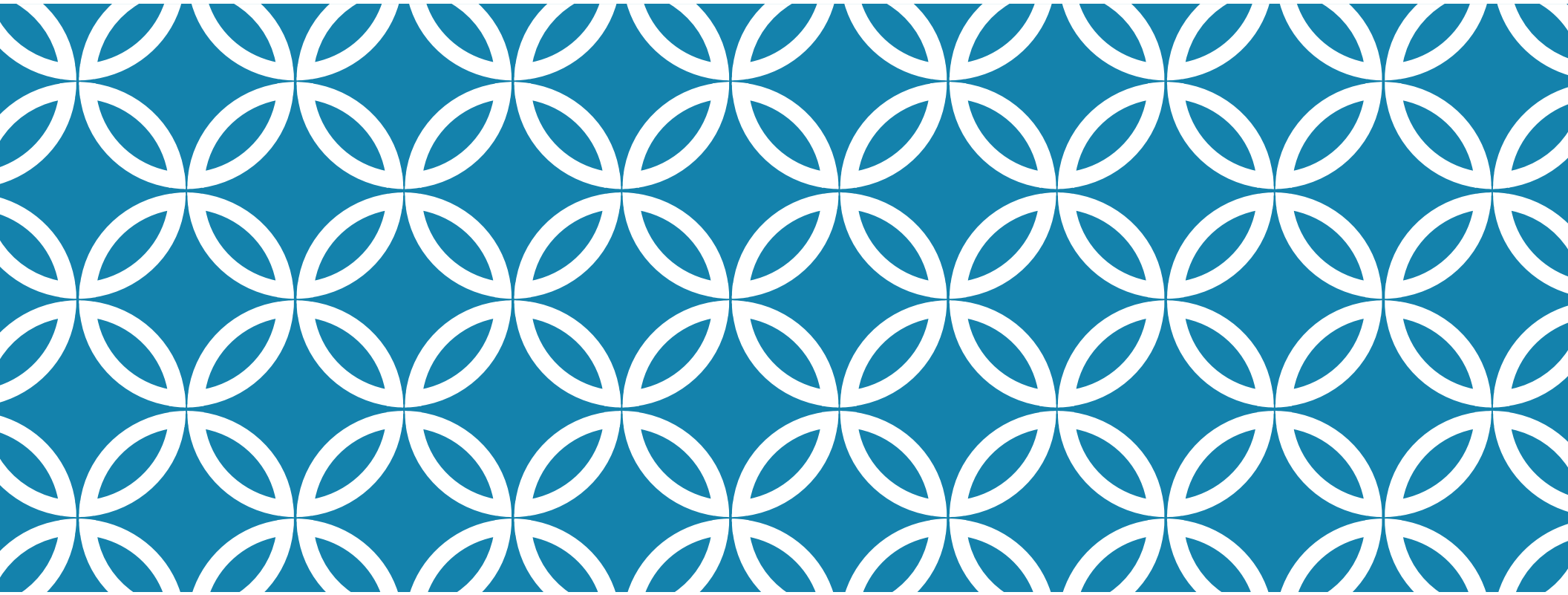


# NETWORKS DESIGN PRINCIPLES

# COMMUNICATION

- Communication between processes over the network, or *interprocess communication* (IPC), is at the heart of distributed systems
- In order for processes to communicate, they need to agree on a set of rules that determine how data is processed and formatted.
- Network protocols specify such rules.
- The protocols are arranged in a stack, where each layer builds on the abstraction provided by the layer below, and lower layers are closer to the hardware.
- When a process sends data to another through the network stack, the data moves from the top layer to the bottom one and vice-versa at the other end.

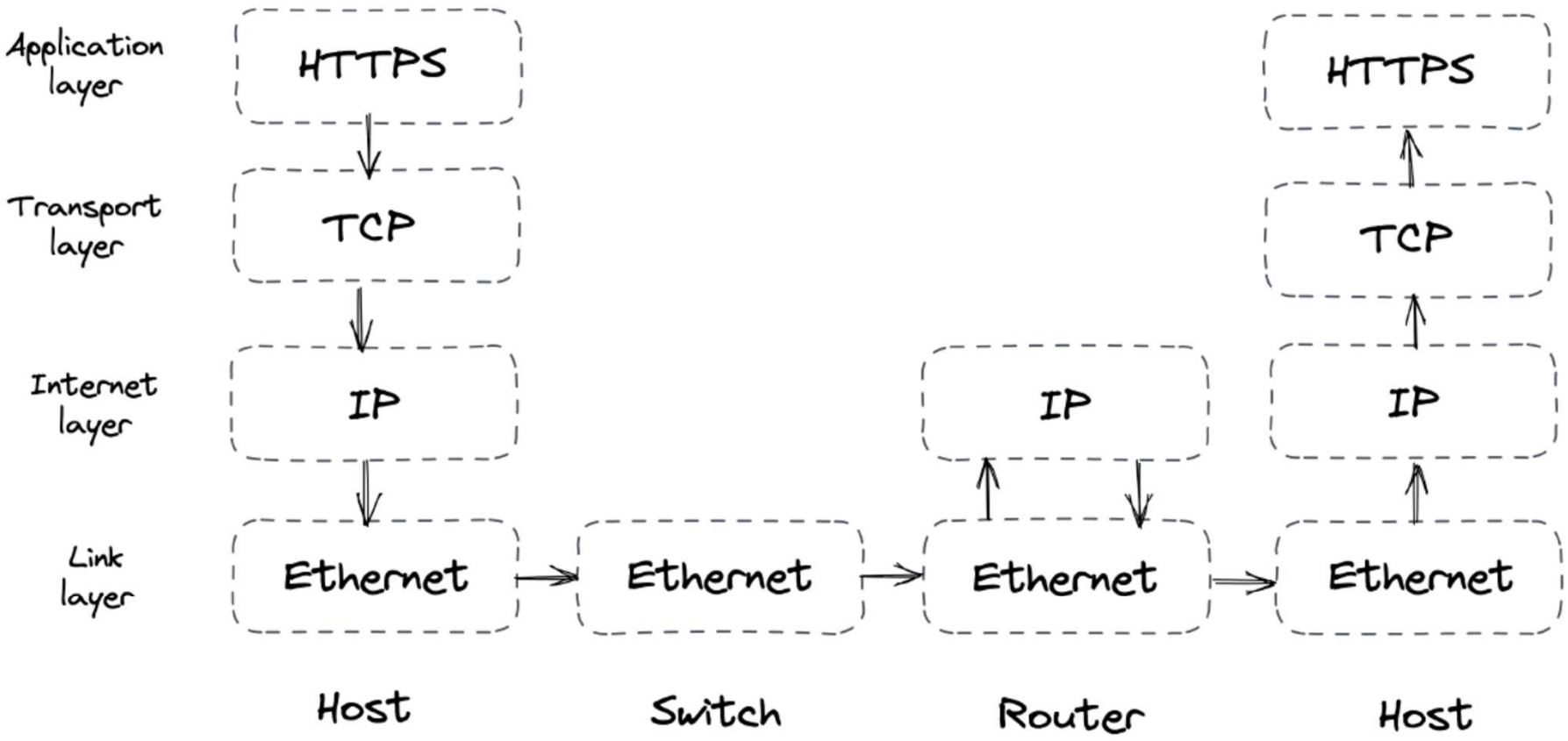
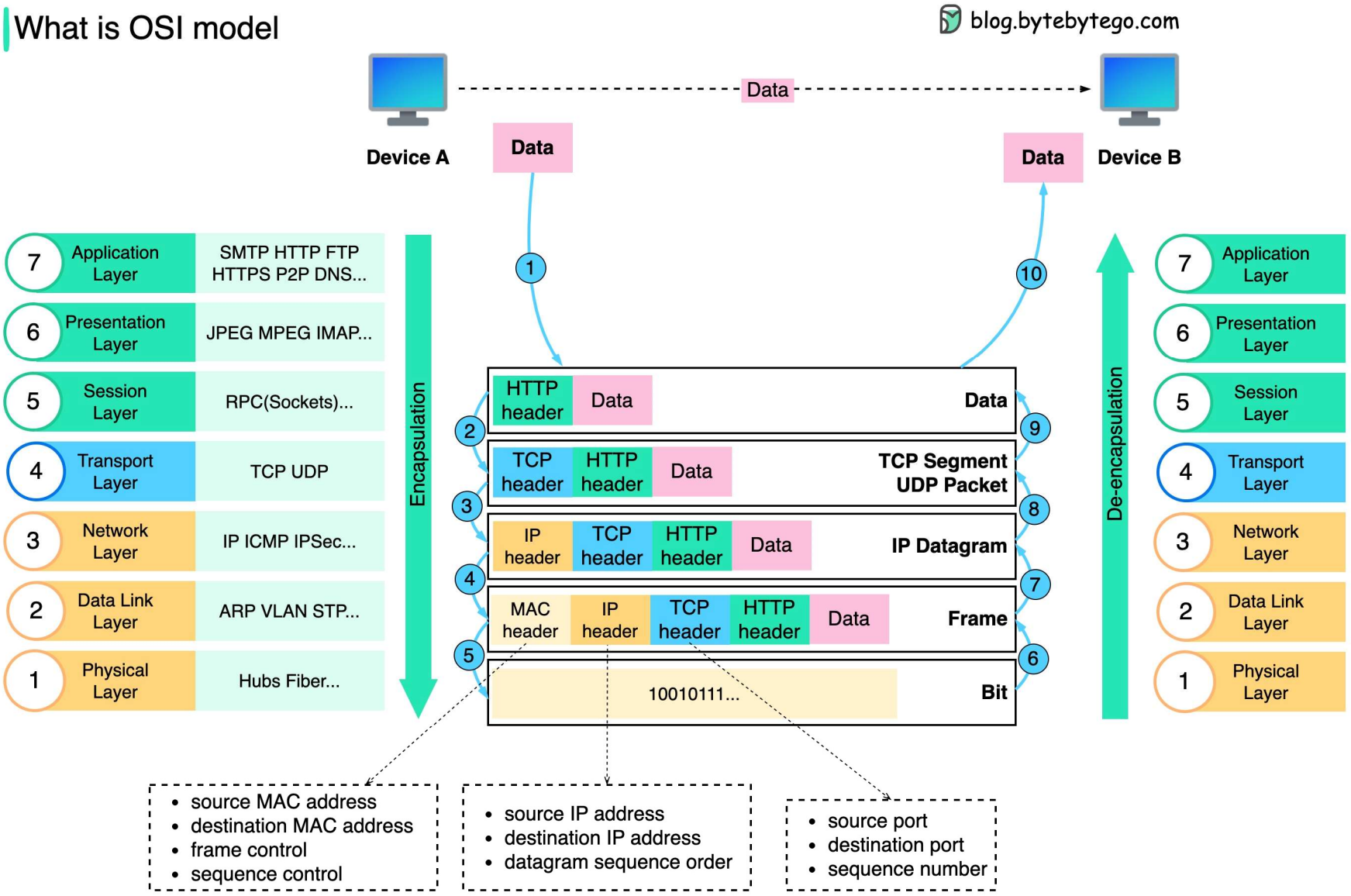



Figure 1.3: Internet protocol suite

# DESCRIPTION OF LAYERS

- The *link layer* consists of network protocols that operate on local network links
  - Ethernet or Wi-Fi, and provides an interface to the underlying network hardware
  - **Switches** operate at this layer and forward Ethernet packets based on their destination MAC addr
- The *internet layer* routes packets from one machine to another across the network. The Internet Protocol (IP) is the core protocol of this layer, which delivers packets on a best-effort basis (i.e., packets can be dropped, duplicated, or corrupted).
  - **Routers** operate at this layer and forward IP packets to the next router along the path to their final destination.
- The *transport layer* transmits data between two processes.
  - To enable multiple processes hosted on the same machine to communicate at the same time, **port numbers** are used to address the processes on either end.
  - Transmission Control Protocol **(TCP)** creates a reliable communication channel on top of IP.
- *Application layer* defines high-level communication protocols, like HTTP or DNS.
  - Typically your applications will target this level of abstraction.

# What is OSI model



# DESIGN LESSONS FROM NETWORKS

- Building reliable abstractions on top of unreliable ones: TCP over IP
  - IP layer can drop or duplicate data or deliver it out of order
- Secure a network connection from prying eyes and malicious agents
  - Secure channel (TLS) on top of a reliable one (TCP)
- Distributed, hierarchical, and eventually consistent key-value store
  - Phone book of the internet (DNS) allows nodes to discover others using names
- Loosely coupled services communicate with each other through APIs
  - Implementation of a RESTful HTTP API

# RELIABLE LINKS

IP Routing needs two components:

A. Addresses (e.g. IPv6 addresses, 128 bits)

B. A mechanism to route packets (e.g. BGP)

What if a router becomes overloaded?

TCP guarantees that a stream of bytes arrives in order without gaps, duplication, or corruption. It also does flow control (protect overload at the network and the receiver).

# TCP SEGMENTS

TCP partitions a byte stream into discrete packets called segments.

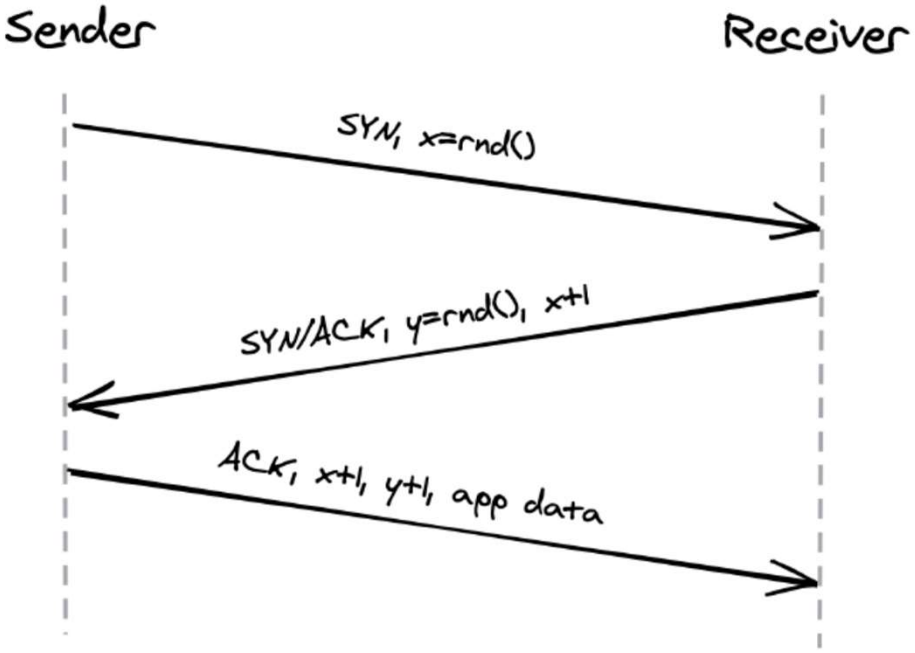
The segments are sequentially numbered, which allows the receiver to detect holes and duplicates.

Every segment sent needs to be acknowledged by the receiver.

When that doesn't happen, a timer fires on the sending side and the segment is retransmitted.

To ensure that the data hasn't been corrupted in transit, the receiver uses a checksum to verify the integrity of a delivered segment.

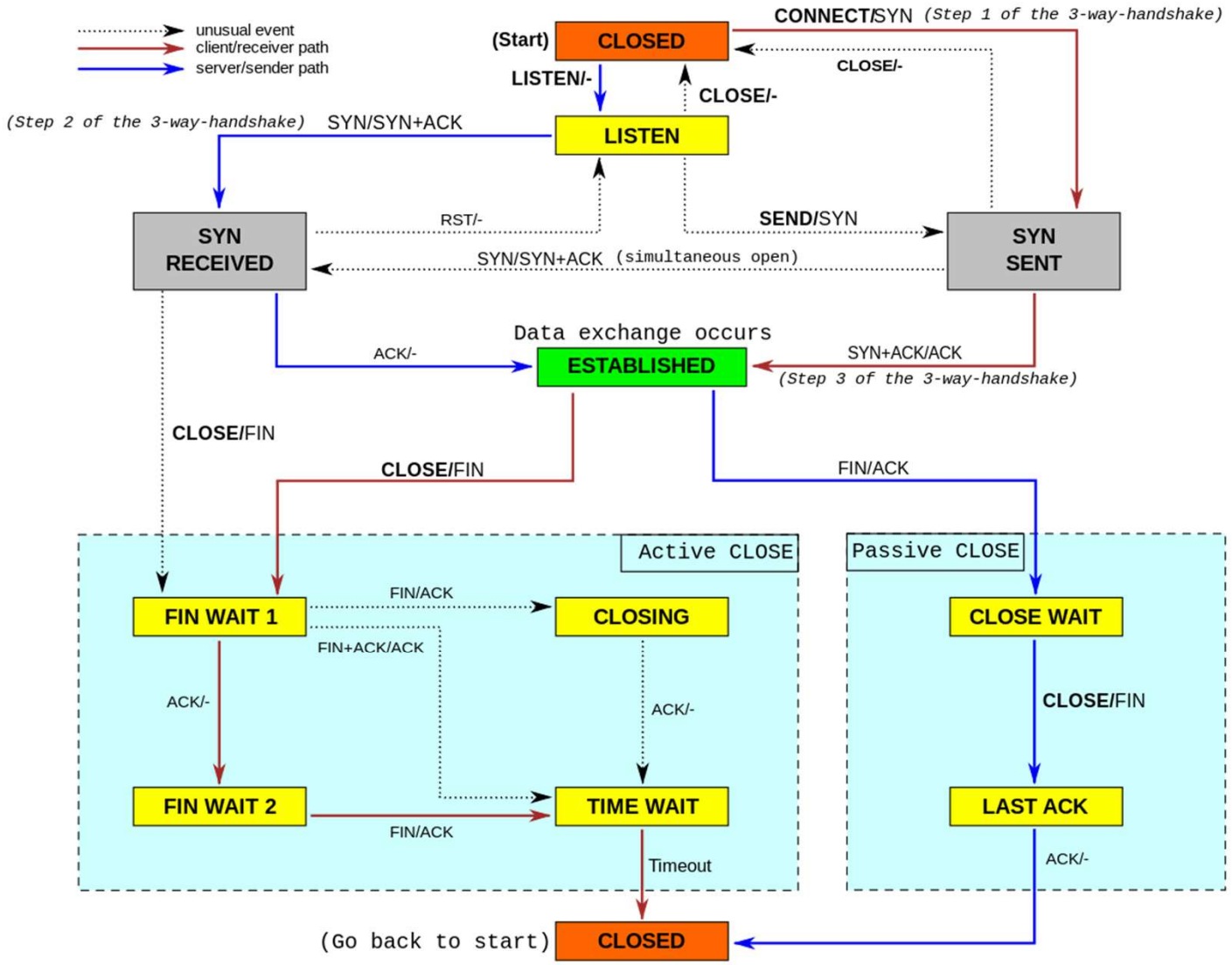
# TCP CONNECTIONS




A server must be listening for connection requests from clients before a connection is established.

Connection states are tracked by a socket.

- The **opening state** in which the connection is being created.
- The **established state** in which the connection is open and data is being transferred.
- The **closing state** in which the connection is being closed.


unusual event
client/receiver path
server/sender path
(Start)
CLOSED
CONNECT/SYN
(Step 1 of the 3-way-handshake)
CLOSE/-
LISTEN/-
CLOSE/-
(Step 2 of the 3-way-handshake)
SYN/SYN+ACK
LISTEN
SYN RECEIVED
RST/-
SEND/SYN
SYN SENT
SYN/SYN+ACK (simultaneous open)
Data exchange occurs
ACK/-
ESTABLISHED
SYN+ACK/ACK
(Step 3 of the 3-way-handshake)
CLOSE/FIN
CLOSE/FIN
FIN/ACK
Active CLOSE
Passive CLOSE
FIN WAIT 1
FIN/ACK
CLOSING
FIN+ACK/ACK
CLOSE WAIT
ACK/-
ACK/-
CLOSE/FIN
FIN WAIT 2
TIME WAIT
LAST ACK
FIN/ACK
Timeout
ACK/-
(Go back to start)
CLOSED

# CONNECTION POOLS

**TCP Cold-start Penalty:** The handshake introduces a full round-trip in which no application data is sent. So until the connection has been opened, the bandwidth is essentially zero. The lower the round trip time is, the faster the connection can be established. Therefore, putting servers closer to the clients helps reduce this cold-start penalty.

**Termination time:** After the data transmission is complete, the connection needs to be closed to release all resources on both ends. This termination phase involves multiple round-trips. If it's likely that another transmission will occur soon, it makes sense to **keep the connection** open to avoid paying the cold-start tax again.

**Closing a socket** doesn't dispose of it immediately as it transitions to a waiting state (*TIME_WAIT*) that lasts several minutes and discards any segments received during the wait. The wait prevents delayed segments from a closed connection from being considered part of a new connection. But if many connections open and close quickly, the number of sockets in the waiting state will continue to increase **until it reaches the maximum number of sockets** that can be open, causing new connection attempts to fail.

This is another reason why processes typically maintain **connection pools** to avoid recreating connections repeatedly.

# FLOW CONTROL

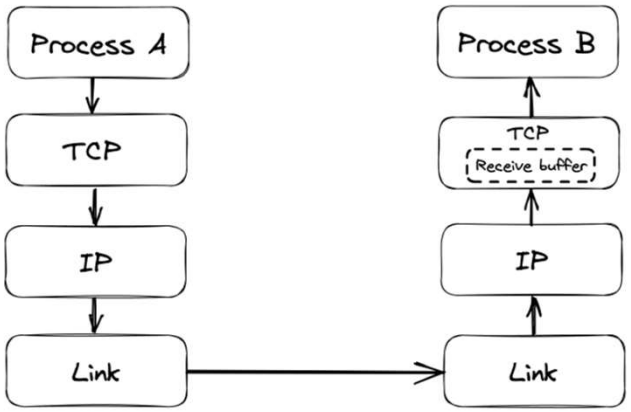



Figure 2.2: The receive buffer stores data that hasn't yet been processed by the destination process.

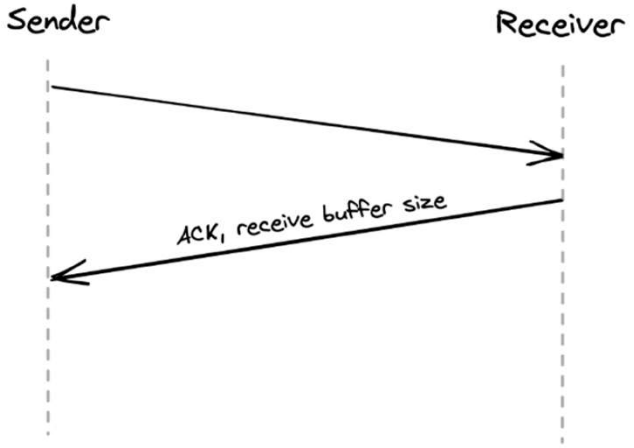



Figure 2.3: The size of the receive buffer is communicated in the headers of acknowledgment segments.

Flow control is a backoff mechanism that TCP implements to prevent the sender from overwhelming the receiver.
The receiver stores incoming TCP segments waiting to be processed by the application into a receive buffer.
But, rather than rate-limiting on an API key or IP address, TCP is rate-limiting on a connection level.

# CONGESTION CONTROL

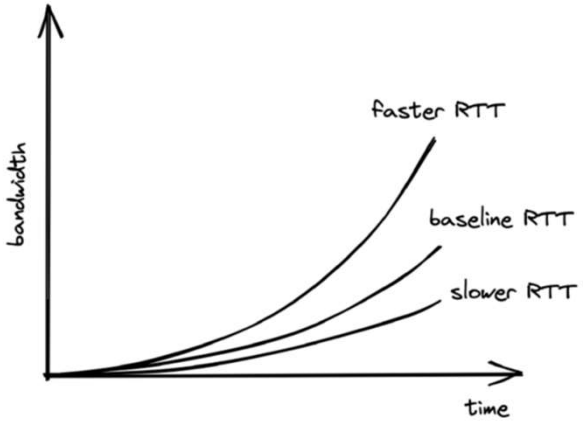
TCP guards not only against overwhelming the receiver, but also against flooding the underlying network.

The sender maintains a so-called *congestion window*, which represents the total **number of outstanding segments that can be sent without an acknowledgment from the other side**.

The smaller the congestion window is, the fewer bytes can be in flight at any given time, and the less bandwidth is utilized.

When a new connection is established, the size of the congestion window is set to a system default.

Then, for every segment acknowledged, the window increases its size exponentially until it reaches an upper limit. This means we can't use the network's full capacity right after a connection is established. The shorter the round-trip time (RTT), the quicker the sender can start utilizing the underlying network's bandwidth.




$$\text{Bandwidth} = \frac{\text{WinSize}}{\text{RTT}}$$

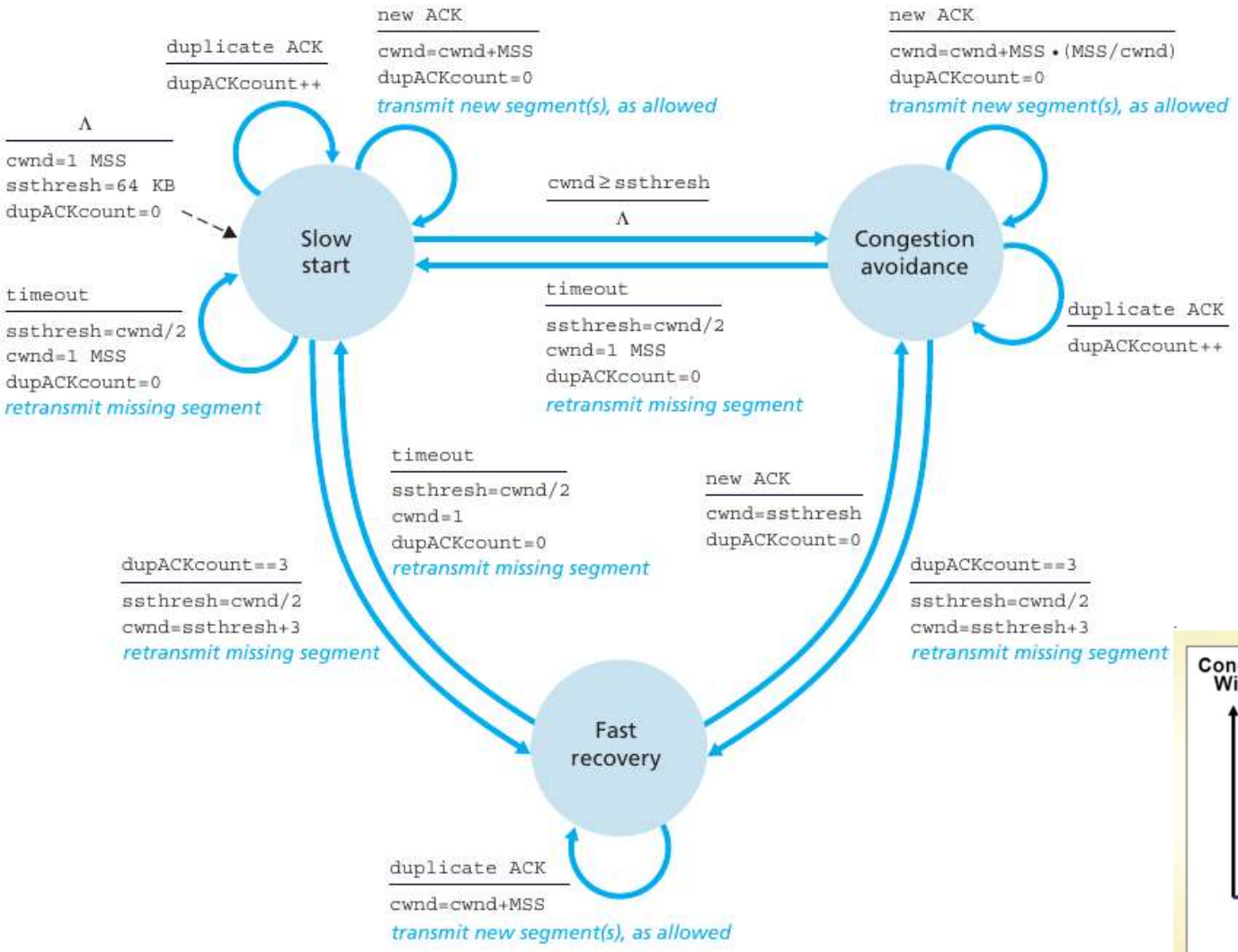
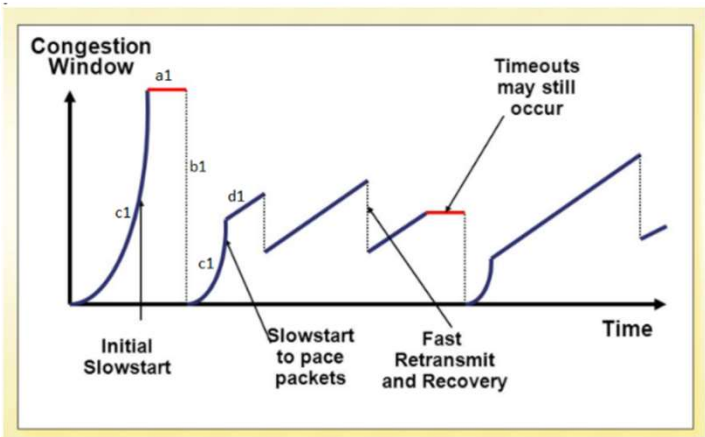



**Figure 3.52** ♦ FSM description of TCP congestion control



# ALTERNATIVES

TCP's reliability and stability come at the price of lower bandwidth and higher latencies than the underlying network can deliver.

If we drop the stability and reliability mechanisms that TCP provides, what we get is a simple protocol named *User Datagram Protocol*[6] (UDP) — a connectionless transport layer protocol that can be used as an alternative to TCP.

# TRANSPORT LAYER SECURITY

TLS runs on top of TCP and encrypts the communication channel so that application layer protocols, like HTTP, can leverage it to communicate securely. In a nutshell, TLS provides *encryption*, *authentication*, and *integrity*.

* Encryption:

When the TLS connection is first opened, the client and the server negotiate a shared encryption secret using *asymmetric encryption*.

First, each party generates a key pair consisting of a private and public key. The processes can then create a shared secret by exchanging their public keys. Although asymmetric encryption is slow and expensive, it's only used to create the shared encryption key.

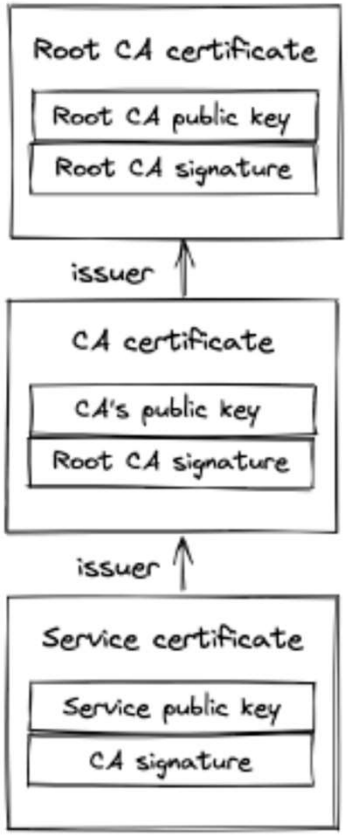
After that, *symmetric encryption* is used, which is fast and cheap. The shared key is periodically renegotiated to minimize the amount of data that can be deciphered if the shared key is broken.

# AUTHENTICATION

TLS implements authentication using digital signatures based on asymmetric cryptography. The server generates a key pair with a private and a public key and shares its public key with the client.

When the server sends a message to the client, it signs it with its private key. The client uses the server's public key to verify that the digital signature was actually signed with the private key.

The problem with this approach is that the client has no idea whether the public key shared by the server is authentic. Hence, the protocol uses certificates to prove the ownership of a public key. A certificate includes information about the owning entity, expiration date, public key, and a digital signature of the third party entity that issued the certificate. The certificate's issuing entity is called a *certificate authority* (CA), which is also represented with a certificate. This creates a chain of certificates that ends with a certificate issued by a root CA, which self-signs its certificate.



# EXPIRED CERTIFICATES

One of the most common mistakes when using TLS is letting a certificate expire.

When that happens, the client won't be able to verify the server's identity, and opening a connection to the remote process will fail.

This can bring an entire application down as clients can no longer connect with it.

For this reason, automation to monitor and auto-renew certificates close to expiration is well worth the investment.

# INTEGRITY

To protect against tampering, TLS verifies the integrity of the data by calculating a message digest. A secure hash function is used to create a message authentication code (HMAC).

When a process receives a message, it recomputes the digest of the message and checks whether it matches the digest included in the message.

If not, then the message has either been corrupted during transmission or has been tampered with. In this case, the message is dropped.

The TLS HMAC protects against data corruption as well, not just tampering. You might be wondering how data can be corrupted if TCP is supposed to guarantee its integrity.

While TCP does use a checksum to protect against data corruption, it's not 100% reliable: it fails to detect errors for roughly 1 in 16 million to 10 billion packets. With packets of 1 KB, this is expected to happen once per 16 GB to 10 TB transmitted.

# HANDSHAKE

When a new TLS connection is established, a handshake between the client and server occurs:

1. The parties agree on the cipher suite to use. A cipher suite specifies the different algorithms that the client and the server intend to use to create a secure channel, like the:
    - key exchange algorithm used to generate shared secrets;
    - signature algorithm used to sign certificates;
    - symmetric encryption algorithm used to encrypt the application data;
    - HMAC algorithm used to guarantee the integrity and authenticity of the application data.

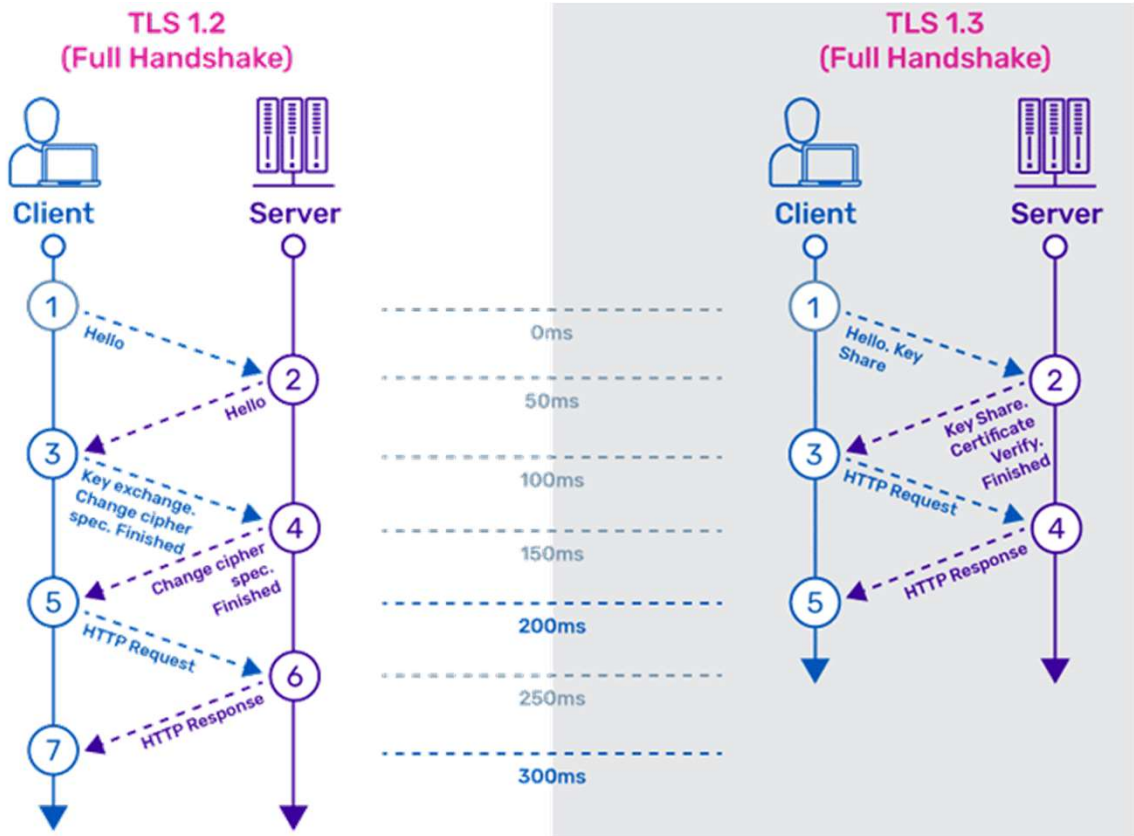
2. The parties use the key exchange algorithm to create a shared secret. The symmetric encryption algorithm uses the shared secret to encrypt communication on the secure channel going forward.

3. The client verifies the certificate provided by the server. The verification process confirms that the server is who it says it is. If the verification is successful, the client can start sending encrypted application data to the server. The server can optionally also verify the client certificate if one is available.

# SETUP TIME

The handshake typically requires 2 round trips with TLS 1.2 and just one with TLS 1.3.

The bottom line is that creating a new connection is not free: yet another reason to put your servers geographically closer to the clients and reuse connections when possible.



# DOMAIN NAME SYSTEM

*Domain Name System* (DNS) — a distributed, hierarchical, and eventually consistent key-value store.

1. Check local cache. Return **if** present, **else** route to DNS Resolver, typically hosted by ISP.

2. **Resolver** iteratively resolves hosts for its clients. It checks its cache. Return **if** present, **else** route to **Root name server** (NS)

3. The **root name server** maps the *top-level domain* (TLD) of the request, i.e., *.com*, to the address of the name server responsible for it.

4. The resolver sends a resolution request for *example.com* to the **TLD name server**.

5. The **TLD name server** maps the *example.com* domain name to the address of the ***authoritative name server*** responsible for the domain.

6. Finally, the resolver queries the **authoritative name server** for *www.example.com*, which returns the IP address of the *www* hostname.

If the query included a subdomain of *example.com*, like *news.example.com*, the authoritative name server would have returned the address of the name server responsible for the subdomain, and an additional request would be required.

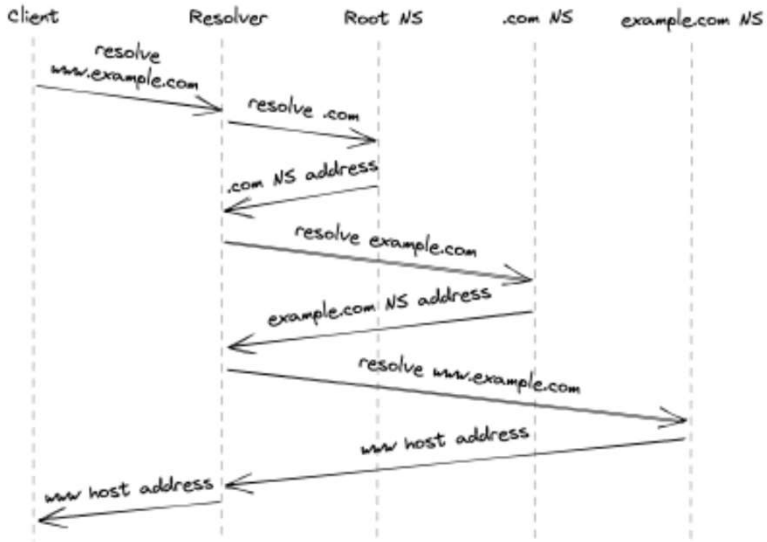



Figure 4.1: DNS resolution process

# CACHING

Caching is used to speed up the resolution process since the mapping of domain names to IP addresses doesn't change often — the browser, operating system, and DNS resolver all use caches internally.

How do these caches know when to expire a record? Every DNS record has a *time to live* (TTL) that informs the cache how long the entry is valid for. But there is no guarantee that clients play nicely and enforce the TTL. So don't be surprised when you change a DNS entry and find out that a small number of clients are still trying to connect to the old address long after the TTL has expired.

Setting a TTL requires making a tradeoff. If you use a long TTL, many clients won't see a change for a long time. But if you set it too short, you increase the load on the name servers and the average response time of requests because clients will have to resolve the hostname more often.

If your name server becomes unavailable for any reason, then the smaller the record's TTL is, the higher the number of clients impacted will be. DNS can easily become a single point of failure — if your DNS name server is down and clients can't find the IP address of your application, they won't be able to connect it. This can lead to massive outages4.

DNS could be a lot more robust to failures if DNS caches would serve stale entries when they can't reach a name server, rather than treating TTLs as time bombs. Since entries rarely change, serving a stale entry is arguably a lot more robust than not serving any entry at all. The principle that a system should continue to function even when a dependency is impaired is also referred to as "static stability"

# API

The server uses an adapter — which defines its application programming interface (API) — to translate messages received from the communication link to interface calls implemented by its business logic.

The communication style between a client and a server can be *direct* or *indirect*, depending on whether the client communicates directly with the server or indirectly through a broker.

Direct communication style called *request-response*, in which a client sends a *request message* to the server, and the server replies with a *response message*. This is similar to a function call but across process boundaries and over the network.

The request and response messages contain data that is serialized in a language-agnostic format. The choice of format determines a message's serialization and deserialization speed, whether it's human-readable, and how hard it is to evolve it over time.

A *textual* format like JSON is self-describing and human-readable, at the expense of increased verbosity and parsing overhead. On the other hand, a binary format like Protocol Buffers is leaner and more performant than a textual one at the expense of human readability.

# SYNCHRONIZATION

When a client sends a request to a server, it can block and wait for the response to arrive, making the communication *synchronous*. Alternatively, it can ask the outbound adapter to invoke a callback when it receives the response, making the communication *asynchronous*.

Synchronous communication is inefficient, as it blocks threads that could be used to do something else. Some languages, like JavaScript, C#, and Go, can completely hide callbacks through language primitives such as async/await. These primitives make writing asynchronous code as straightforward as writing synchronous code.

# IPC TECHNOLOGIES

Commonly used IPC technologies for request-response interactions are HTTP and gRPC. Typically, internal APIs used for server-to-server communications within an organization are implemented with a high-performance RPC framework like gRPC.

In contrast, external APIs available to the public tend to be based on HTTP, since web browsers can easily make HTTP requests via JavaScript code.

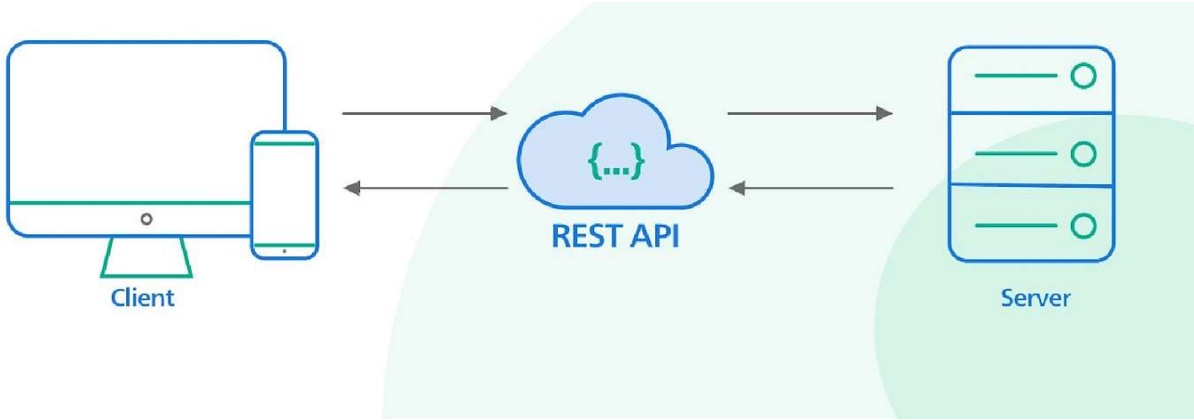
A popular set of design principles for designing elegant and scalable HTTP APIs is representational state transfer (REST), and an API based on these principles is said to be RESTful.

For example, these principles include that:

- requests are stateless, and therefore each request contains all the necessary information required to process it;

- responses are implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, the client can reuse the response for a later, equivalent request.

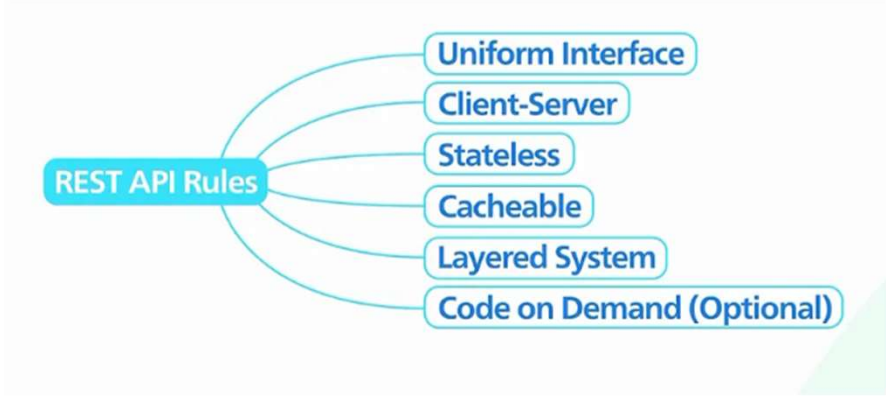
# REST APIS




The common API standard used by most mobile and web applications to talk to the servers is called REST. It stands for REpresentational State Transfer.

REST is not a specification. It is a loose set of rules that has been the de facto standard for building web API since the early 2000s

An API that follows the REST standard is called a RESTful API. Some real-life examples are Twilio, Stripe, and Google Maps.



# HTTP

*HTTP* is a request-response protocol used to encode and transport information between a client and a server. In an *HTTP transaction*, the client sends a *request message* to the server's API endpoint, and the server replies back with a *response message*.

In HTTP 1.1, a message is a textual block of data that contains a start line, a set of headers, and an optional body:

- In a request message, the *start line* indicates what the request is for, and in a response message, it indicates whether the request was successful or not.
- The *headers* are key-value pairs with metadata that describes the message.
- The message *body* is a container for data.

HTTP is a stateless protocol, which means that everything needed by a server to process a request needs to be specified within the request itself, without context from previous requests. HTTP uses TCP for the reliability guarantees. When it runs on top of TLS, it's also referred to as HTTPS.

# ISSUES WITH HTTP1.1

HOL Blocking: Transactions need to be serialized

For example, a browser that needs to fetch several images to render an HTML page has to download them one at a time, which can be very inefficient.

Possible solutions:

A. Pipelining: A single slow response blocks everything after it

B. Create multiple connections: Resource consumption (memory and sockets)

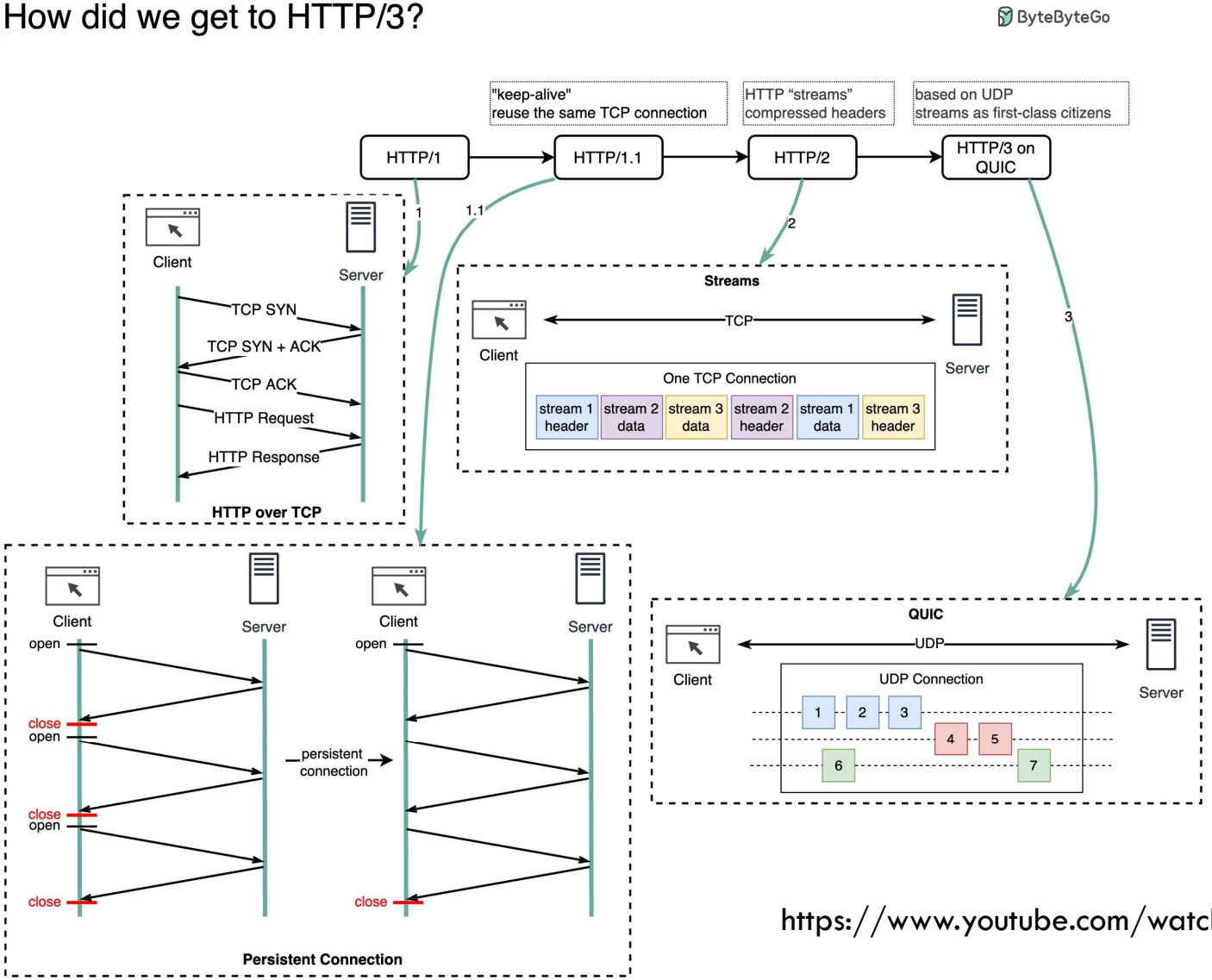
HTTP 2: Uses a binary protocol rather than a textual one, allowing it to multiplex multiple concurrent request/response transactions (streams) on the same connection.

# HTTP 3

HTTP 3 is the latest iteration of the HTTP standard, which is based on UDP and implements its own transport protocol to address some of TCP's shortcomings.

For example, with HTTP 2, a packet loss over the TCP connection blocks all streams (HOL), but with HTTP 3 a packet loss interrupts only one stream, not all of them.

# How did we get to HTTP/3?




https://www.youtube.com/watch?v=rIN4F1oyaRM

# HOSTING RESOURCES

Suppose we would like to implement a service to manage the product catalog of an e-commerce application. The service must allow customers to browse the catalog and administrators to create, update, or delete products. Although that sounds simple, in order to expose this service via HTTP, we first need to understand how to model APIs with HTTP.

An HTTP server hosts resources, where a *resource* can be a physical or abstract entity, like a document, an image, or a collection of other resources. A URL identifies a resource by describing its location on the server.

In our catalog service, the collection of products is a type of resource, which could be accessed with a URL like *https://www.example.com/products?sort=price*, where:

- *https* is the protocol;
- *www.example.com* is the hostname;
- *products* is the name of the resource;
- *?sort=price* is the query string, which contains additional parameters that affect how the service handles the request; in this case, the sort order of the list of products returned in the response.

The URL without the query string is also referred to as the API's */products* endpoint.

# MODELING RELATIONSHIPS B/W RESOURCES

URLs can also model relationships between resources.

For example, since a product is a resource that belongs to the collection of products, the product with the unique identifier 42 could have the following relative URL: */products/42*.

And if the product also has a list of reviews associated with it, we could append its resource name to the product's URL, i.e., */products/42/reviews*.

However, the API becomes more complex as the nesting of resources increases, so it's a balancing act.

Naming resources is only one part of the equation; we also have to serialize the resources on the wire when they are transmitted in the body of request and response messages. When a client sends a request to get a resource, it adds specific headers to the message to describe the preferred representation for the resource. The server uses these headers to pick the most appropriate representation for the response.

# REST INTERNALS

A RESTful API organizes resources into a set of unique URIs or Uniform Resource Identifiers.

**URIs**

```
https://example.com/api/v3/products
https://example.com/api/v3/users
```

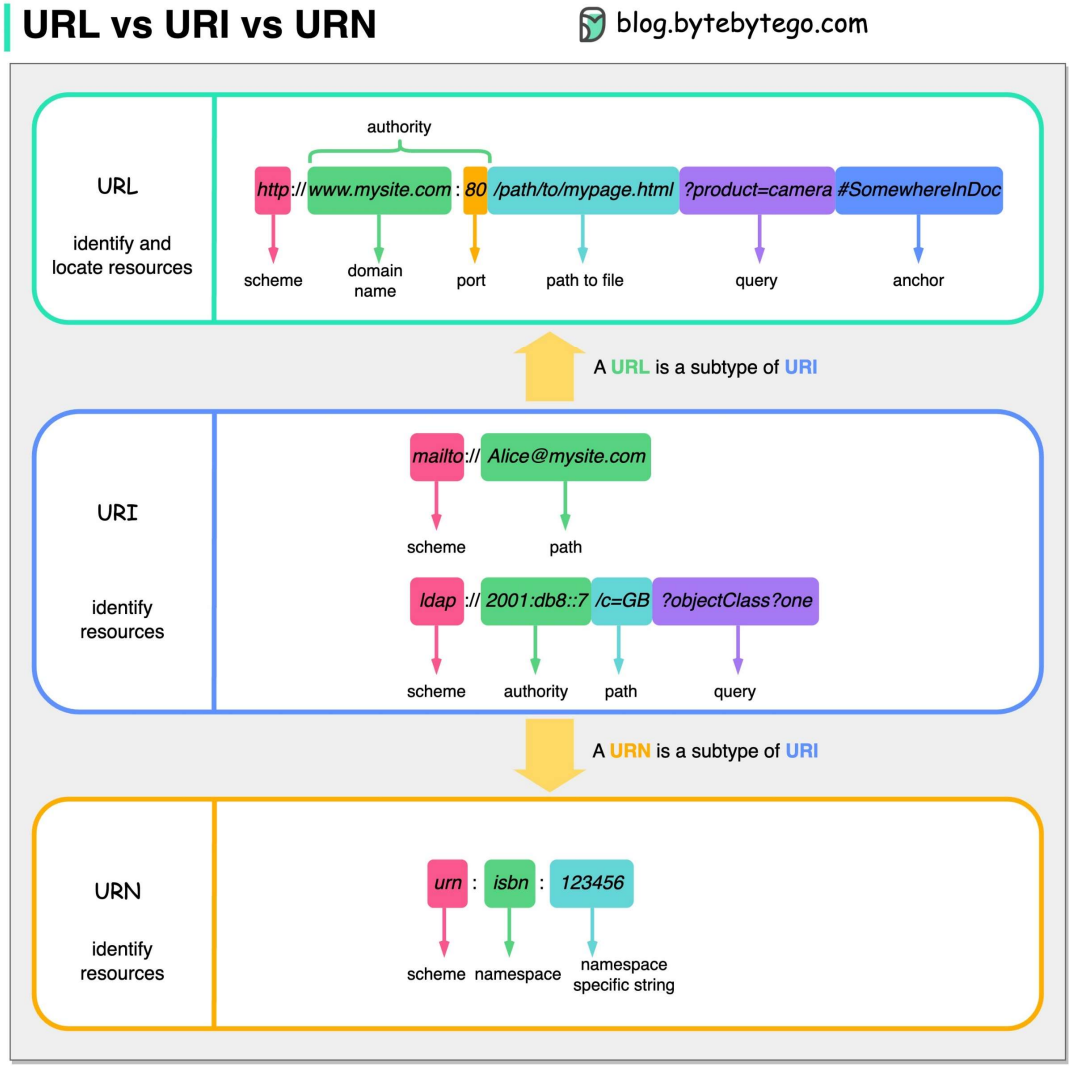
The resources should be grouped by noun and not verb. An API to get all products should be

**Correct**

```
https://example.com/api/v3/products
```

**Wrong**

```
https://example.com/api/v3/getAllProducts
```



# SERIALIZATION

Generally, in HTTP APIs, JSON is used to represent non-binary resources.

```
{
  "id": 42,
  "category": "Laptop",
  "price": 999
}
```

# REQUEST METHODS

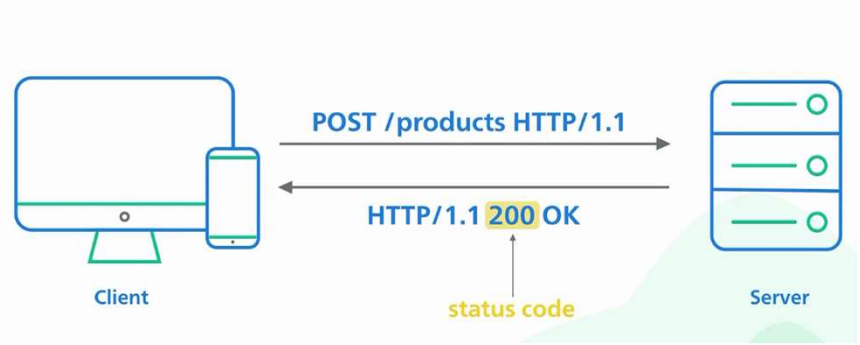
HTTP requests can create, read, update, and delete (CRUD) resources using request *methods*. When a client makes a request to a server for a particular resource, it specifies which method to use.
You can think of a request method as the verb or action to use on a resource.
The most commonly used methods are *POST*, *GET*, *PUT*, and *DELETE*.

For example, the API of our catalog service could be defined as follows:

- *POST /products* — Create a new product and return the URL of the new resource.
- *GET /products* — Retrieve a list of products. The query string can be used to filter, paginate, and sort the collection.
- *GET /products/42* — Retrieve product 42.
- *PUT /products/42* — Update product 42.
- *DELETE /products/42* — Delete product 42.

# EXAMPLE

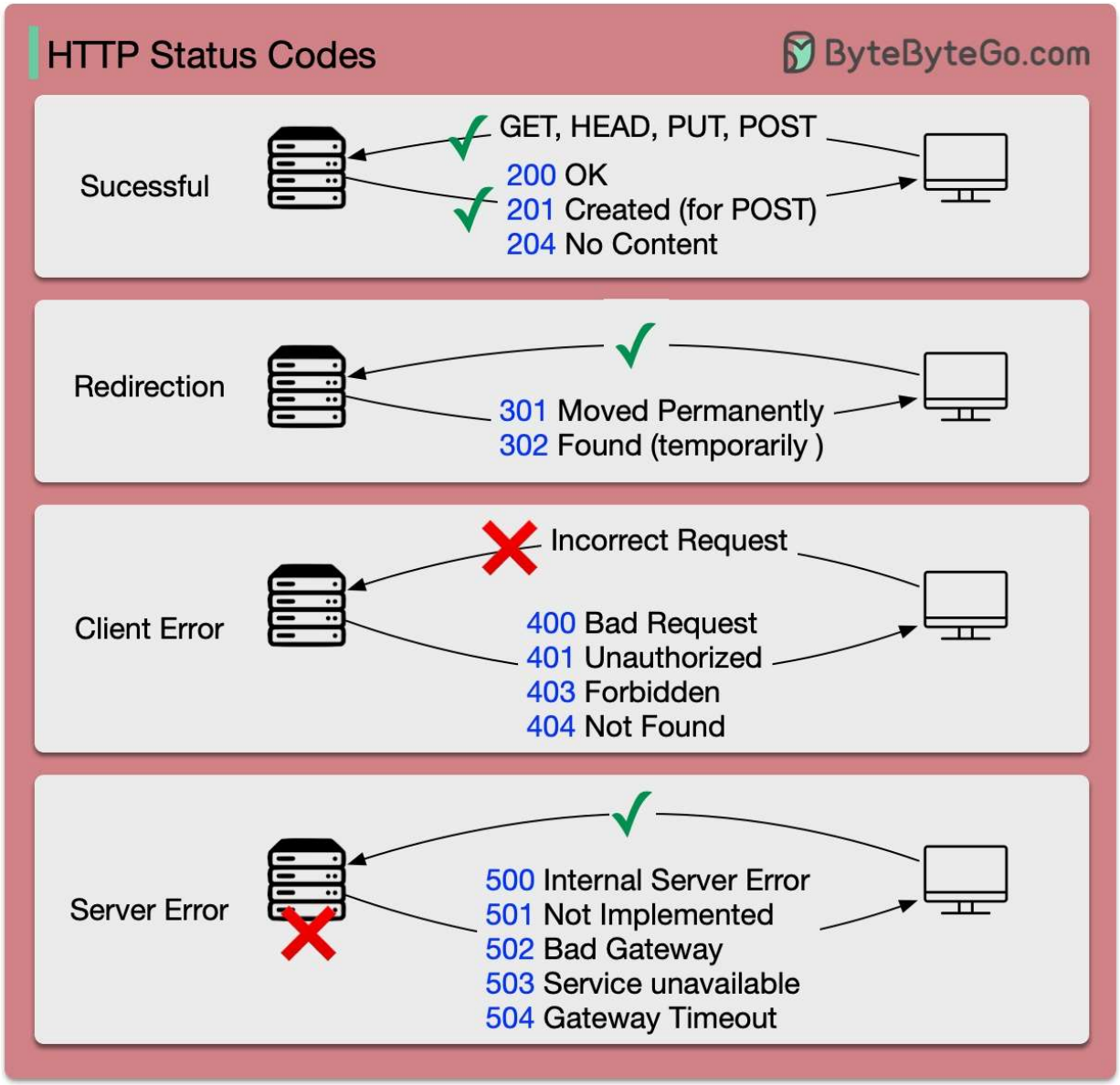

POST /products HTTP/1.1
Accept: application/json
JSON:
{
"customer": "Jason Sweet"
"quantity": 1,
"price": 18.00
}
Client
Server


POST /products HTTP/1.1
HTTP/1.1 200 OK
status code
Client
Server

# CATEGORIES OF REQUEST METHODS

Request methods can be categorized based on whether they are safe and whether they are idempotent. A *safe* method should not have any visible side effects and can safely be cached. An *idempotent* method can be executed multiple times, and the end result should be the same as if it was executed just a single time. Idempotency is a crucial aspect of APIs

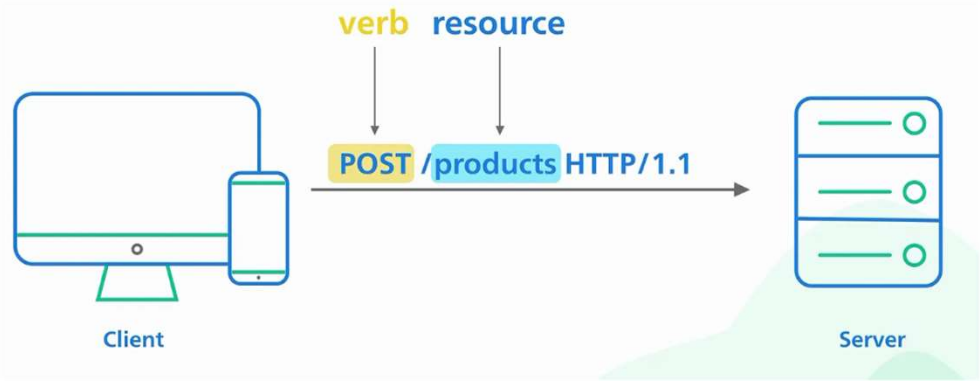
| Method | Safe | Idempotent |
|---|---|---|
| POST | No | No |
| GET | Yes | Yes |
| PUT | No | Yes |
| DELETE | No | Yes |


HTTP Status Codes
ByteByteGo.com
Sucessful
GET, HEAD, PUT, POST
200 OK
201 Created (for POST)
204 No Content
Redirection
301 Moved Permanently
302 Found (temporarily )
Client Error
Incorrect Request
400 Bad Request
401 Unauthorized
403 Forbidden
404 Not Found
Server Error
500 Internal Server Error
501 Not Implemented
502 Bad Gateway
503 Service unavailable
504 Gateway Timeout

A client interacts with a resource by making a request to the endpoint for the resource over HTTP. The request has a very specific format.

**POST /products HTTP/1.1**

The line contains the URI for the resource we'd like to access. The URI is preceded by an HTTP verb which tells the server what we want to do with the resource.



# IMPORTANT ASPECTS

**Stateless:** A REST implementation should be stateless. It means that the two parties don't need to store any information about each other and every request and response is independent from all others.

**Pagination:** If an API endpoint returns a huge amount of data, use pagination. A common pagination scheme uses limit and offset.

**Versioning:** Versioning allows an implementation to provide backward compatibility so that if we introduce breaking changes from one version to another, consumers get enough time to move to the next version.

There are many ways to version an API. The most straightforward is to prefix the version before the resource on the URI.

# ADAPTERS

```
interface CatalogService
{
  List<Product> GetProducts(...);
  Product GetProduct(...);
  void AddProduct(...);
  void DeleteProduct(...);
  void UpdateProduct(...)
}
```

An adapter that handles HTTP requests by calling the business logic of the catalog service. For example, suppose the service is defined by the following interface above.

So when the HTTP adapter receives a *GET /products* request to retrieve the list of all products, it will invoke the *GetProducts(…)* method and convert the result into an HTTP response.

We can generate a skeleton of the HTTP adapter by defining the API of the service with an *interface definition language* (IDL). An IDL is a language-independent definition of the API that can be used to generate boilerplate code for the server-side adapter and client-side software development kits (SDKs) in your languages of choice.

The OpenAPI is one of the most popular IDLs for RESTful HTTP APIs. With it, we can formally describe the API in a YAML document, including the available endpoints, supported request methods, and response status codes for each endpoint, and the schema of the resources' JSON representation.

# OPENAPI

```yaml
openapi: 3.0.0
info:
  version: "1.0.0"
  title: Catalog Service API

paths:
  /products:
    get:
      summary: List products
      parameters:
        - in: query
          name: sort
          required: false
          schema:
            type: string
      responses:
        "200":
          description: list of products in catalog
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: "#/components/schemas/ProductItem"
        "400":
          description: bad input

components:
  schemas:
    ProductItem:
      type: object
      required:
        - id
        - name
        - category
      properties:
        id:
          type: number
        name:
          type: string
        category:
          type: string
```

With this definition, we can then run a tool to generate the API's documentation, boilerplate adapters, and client SDKs.

# API EVOLUTION

The last thing we want to do when evolving an API is to introduce a breaking change that requires all clients to be modified at once, some of which we might have no control over.

There are two types of changes that can break compatibility, one at **the endpoint level** and another at the **message level.**

For example, if we were to change the */products* endpoint to */new-products*, it would obviously break clients that haven't been updated to support the new endpoint. The same applies when making a previously optional query parameter mandatory.

Changing the schema of request or response messages in a backward-incompatible way can also wreak havoc.

For example, changing the type of the *category* property in the *Product* schema from string to number is a breaking change that would cause the old deserialization logic to blow up in clients.

Similar arguments can be made for messages represented with other serialization formats, like Protocol Buffers.

REST APIs should be versioned to support breaking changes, e.g., by prefixing a version number in the URLs (*/v1/products/*).

However, as a general rule of thumb, APIs should evolve in a backward-compatible way unless there is a very good reason. Although backward-compatible APIs tend not to be particularly elegant, they are practical.

# IDEMPOTENCY

When an API request times out, the client has no idea whether the server actually received the request or not. For example, the server could have processed the request and crashed right before sending a response back to the client.

An effective way for clients to deal with transient failures such as these is to retry the request one or more times until they get a response back. Some HTTP request methods (e.g., PUT, DELETE) are considered inherently idempotent as the effect of executing multiple identical requests is identical to executing only one request.

For example, if the server processes the same PUT request for the same resource twice in a row, the end effect would be the same as if the PUT request was executed only once.

But what about requests that are not inherently idempotent? For example, suppose a client issues a POST request to add a new product to the catalog service. If the request times out, the client has no way of knowing whether the request succeeded or not. If the request succeeds, retrying it will create two identical products, which is not what the client intended.

In order to deal with this case, the client might have to implement some kind of reconciliation logic that checks for duplicate products and removes them when the request is retried. You can see how this introduces a lot of complexity for the client. Instead of pushing this complexity to the client, a better solution would be for the server to create the **product only once by making the POST request idempotent**, so that no matter how many times that specific request is retried, it will appear as if it only executed once.

# SERVER SIDE DUPLICATE CHECK

For the server to detect that a request is a duplicate, it needs to be decorated with an idempotency key — a unique identifier (e.g., a UUID). The identifier could be part of a header, like *Idempotency- Key* in Stripe's API18. For the server to detect duplicates, it needs to remember all the request identifiers it has seen by storing them in a database.

When a request comes in, the server checks the database to see if the request ID is already present. If it's not there, it adds the request identifier to the database and executes the request. Request identifiers don't have to be stored indefinitely, and they can be purged after some time.

Suppose the server adds the request identifier to the database and crashes before executing the request. In that case, any future retry won't have any effect because the server will think it has already executed it. So what we really want is for the request to be handled atomically: either the server processes the request successfully and adds the request identifier to the database, or it fails to process it without storing the request identifier.

# EDGE CASES

If the request identifiers and the resources managed by the server are stored in the same database, we can guarantee atomicity with ACID transactions. In other words, we can wrap the product creation and request identifier log within the same database transaction in the POST handler. However, if the handler needs to make external calls to other services to handle the request, the implementation becomes a lot more challenging, since it requires some form of coordination.

Now, assuming the server can detect a duplicate request, how should it be handled?

In our example, the server could respond to the client with a status code that signals that the product already exists. But then the client would have to deal with this case differently than if it had received a successful response for the first POST request (*201 Created*). So ideally, the server should return the same response that it would have returned for the very first request.

So far, we have only considered a single client. Now, imagine the following scenario:

1. Client A sends a request to create a new product. Although the request succeeds, the client doesn't receive a timely response.

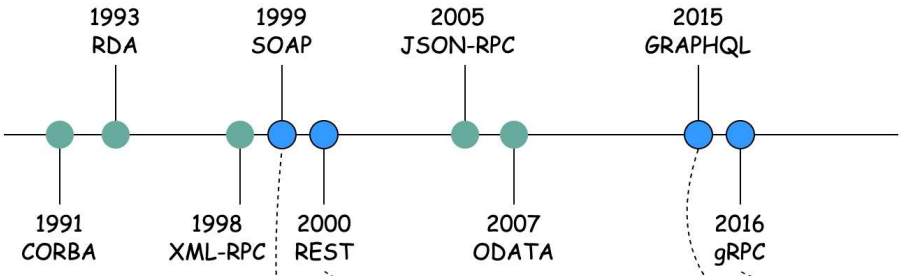
2. Client B deletes the newly created product.

3. Client A retries the original creation request.

How should the server deal with the request in step 3? From client's A perspective, it would be less surprising to receive the original creation response than some strange error mentioning that the resource created with that specific request identifier has been deleted. When in doubt, it's helpful to follow the principle of least astonishment.

# API Architectural Styles Comparison

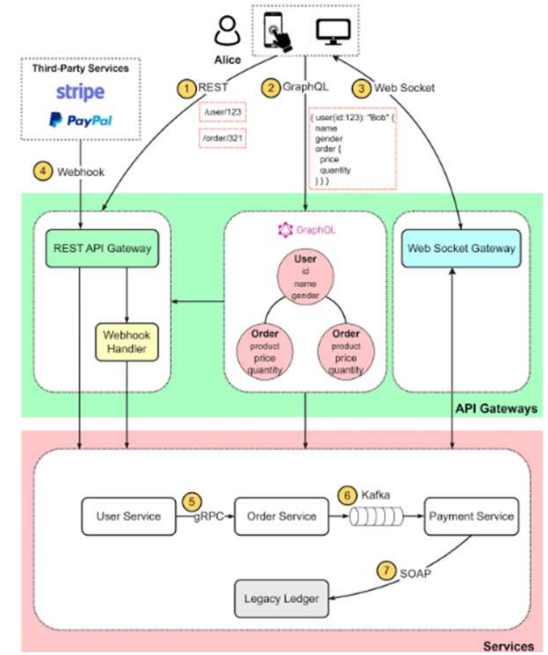
Source: altexsoft




| | SOAP (Simple Object Access Protocol) | REST (REpresentational State Transfer) | GraphQL | RPC (Remote Procedure Call) |
|---|---|---|---|---|
| Organized in terms of | enveloped message structure | compliance with six architectural constraints | schema & type system | local procedure call |
| Format | XML only | XML, JSON, HTML, plain text | JSON | JSON, XML, Protobuf, Thrift, FlatBuffers |
| Learning curve | Difficult | Easy | Medium | Easy |
| Community | Small | Large | Growing | Large |
| Use cases | - payment gateways<br>- identity management<br>- CRM solutions<br>- financial and telecommunication services<br>- legacy system support | - public APIs<br>- simple resource-driven apps | - mobile APIs<br>- complex systems<br>- micro-services | - command and action-oriented APIs<br>- high performance communication in massive micro-services systems |

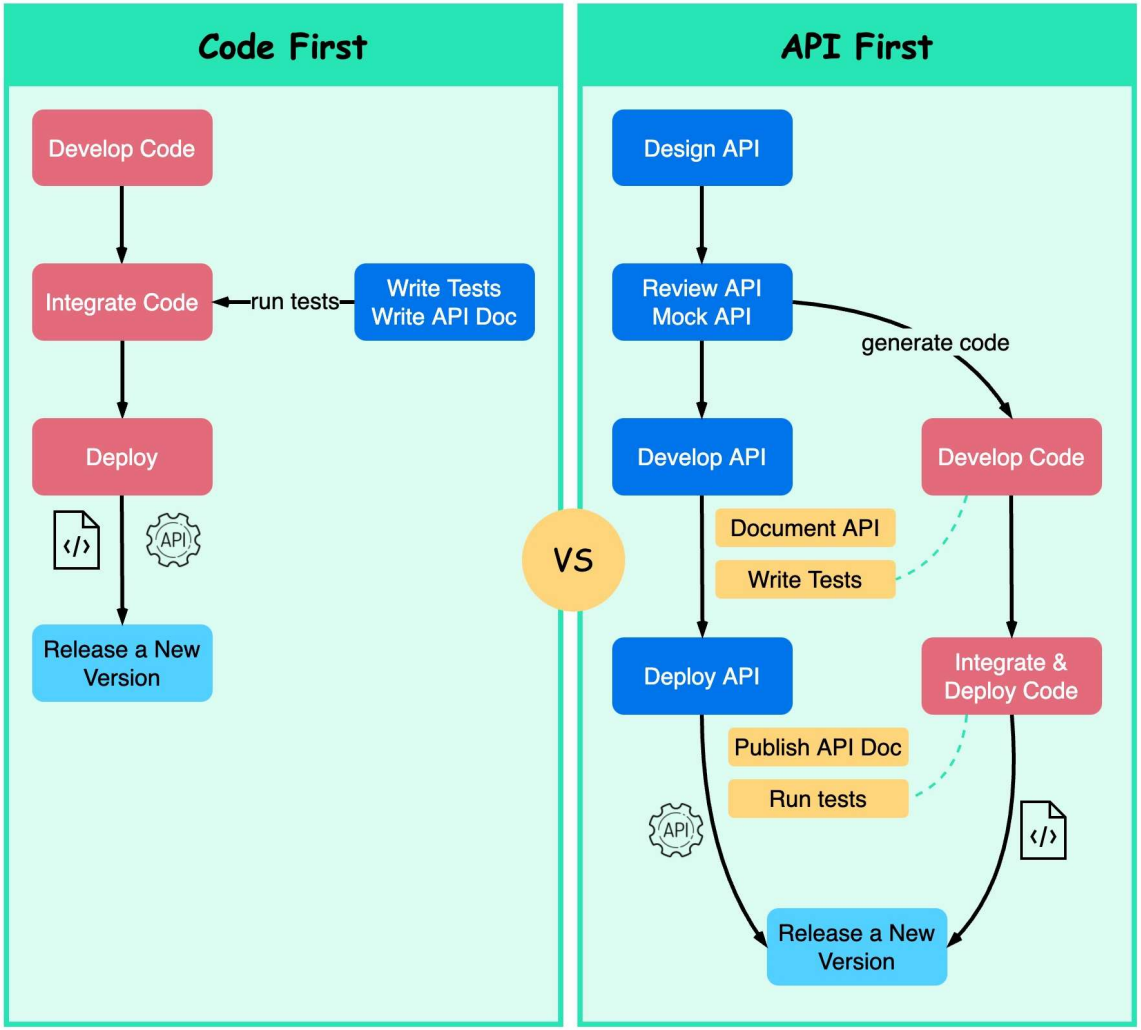
## Jeff Bezos' API Mandate Paraphrased (2002)

- *Data and Capabilities must be exposed through APIs.*
- *Team Communications must be through APIs.*
- *There can be no side channels/shortcuts.*
- *Technology choice is secondary.*
- *APIs must be externalizable.*



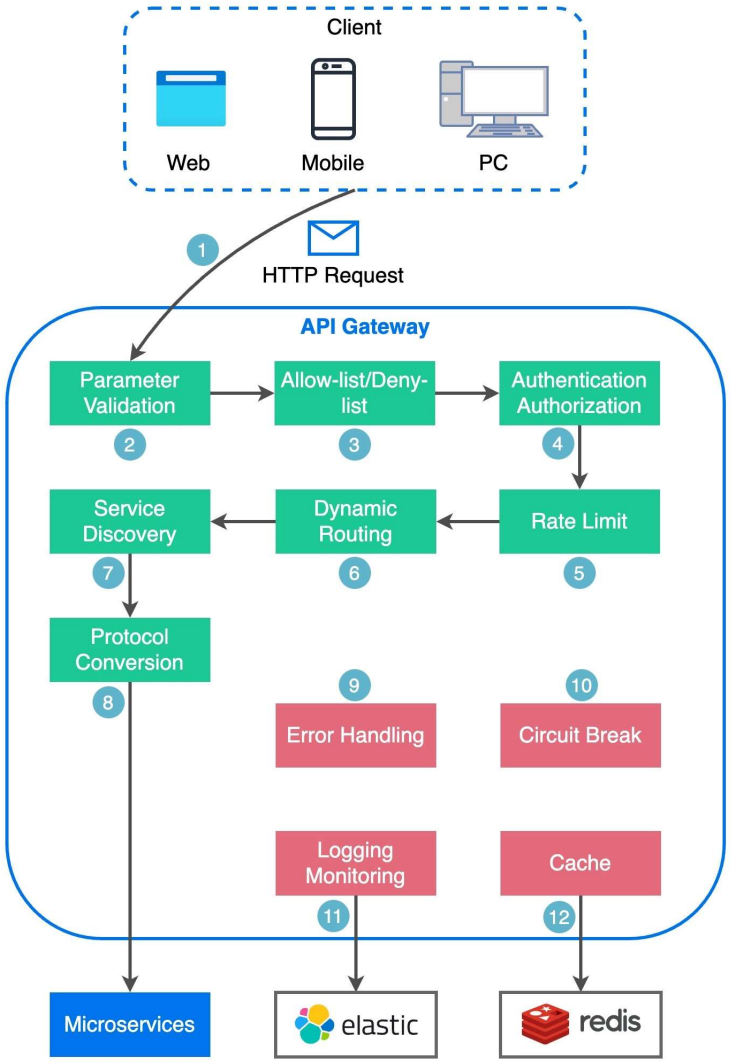

# Code First v.s API First Development

blog.bytebytego.com



# What does API Gateway do?

blog.bytebytego.com

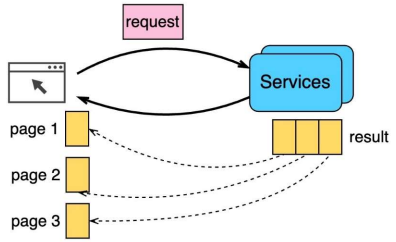
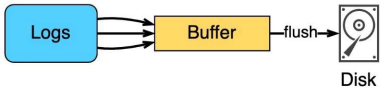
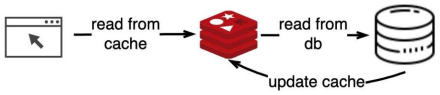
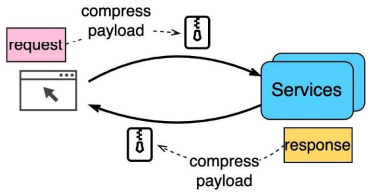
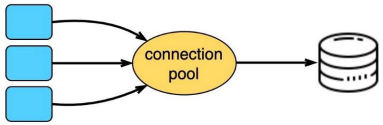


# Design Effective & Safe APIs

blog.bytebytego.com


| | | |
|---|---|---|
| Use resource names (nouns) | GET /querycarts/123 | GET /carts/123 |
| Use plurals | GET /cart/123 | GET /carts/123 |
| Idempotency | POST /carts | POST /carts<br>{requestId: 4321} |
| Use versioning | GET /carts/v1/123 | GET /v1/carts/123 |
| Query after soft deletion | GET /carts | GET /carts?<br>includeDeleted=true |
| Pagination | GET /carts | GET /carts?<br>pageSize=xx&pageToken=xx |
| Sorting | GET /items | GET /items?<br>sort_by=time |
| Filtering | GET /items | GET /items?<br>filter=color:red |
| Secure Access | X-API-KEY=xxx | X-API-KEY = xxx<br>X-EXPIRY = xxx<br>X-REQUEST-SIGNATURE = xxx<br>hmac(URL + QueryString + Expiry + Body) |
| Resource cross reference | GET /carts/123?<br>item=321 | GET<br>/carts/123/items/321 |
| Add an item to a cart | POST /carts/123?<br>addItem=321 | POST<br>/carts/123/items:add<br>{ itemId: "items/321" } |
| Rate limit | No rate limit - DDos | Design rate limiting rules based on IP, user, action group etc |

# How to Improve API Performance?

| | | |
|---|---|---|
| **Pagination** |  | • an ordinal numbering of pages<br>• handles a large number of results |
| **Async Logging** |  | • send logs to a lock-free ring buffer and return<br>• flush to the disk periodically<br>• higher throughput and lower latency |
| **Caching** |  | • store frequently used data in the cache instead of database<br>• query the database when there is a cache miss |
| **Payload Compression** |  | • reduce the data size to speed up the download and upload |
| **Connection Pool** |  | • opening and closing DB connections add significant overhead<br>• a connection pool maintains a number of open connections for applications to reuse |

Reference: Rapid API

# GRPC

Step 1: A REST call is made from the client. The request body is usually in JSON format.

Steps 2 - 4: The order service (gRPC client) receives the REST call, transforms it, and makes an RPC call to the payment service. gPRC encodes the **client stub** into a binary format and sends it to the low-level transport layer.

Step 5: gRPC sends the packets over the network via HTTP2. Because of binary encoding and network optimizations, gRPC is said to be 5X faster than JSON.

Steps 6 - 8: The payment service (gRPC server) receives the packets from the network, decodes them, and invokes the server application.

Steps 9 - 11: The result is returned from the server application, and gets encoded and sent to the transport layer.

Steps 12 - 14: The order service receives the packets, decodes them, and sends the result to the client application.