Introduction to a CDN

Learn about CDNs, and formalize the requirements for a CDN design.

We'll cover the following

Proposed solution

The solution to all the problems discussed in the previous lesson is the **content delivery network** (CDN). A CDN is a group of geographically distributed proxy servers. A **proxy server** is an intermediate server between a client and the origin server. The proxy servers are placed on the network edge. As the network edge is close to the end users, the placement of proxy servers helps quickly deliver the content to the end users by reducing latency and saving bandwidth. A CDN has added intelligence on top of being a simple proxy server.

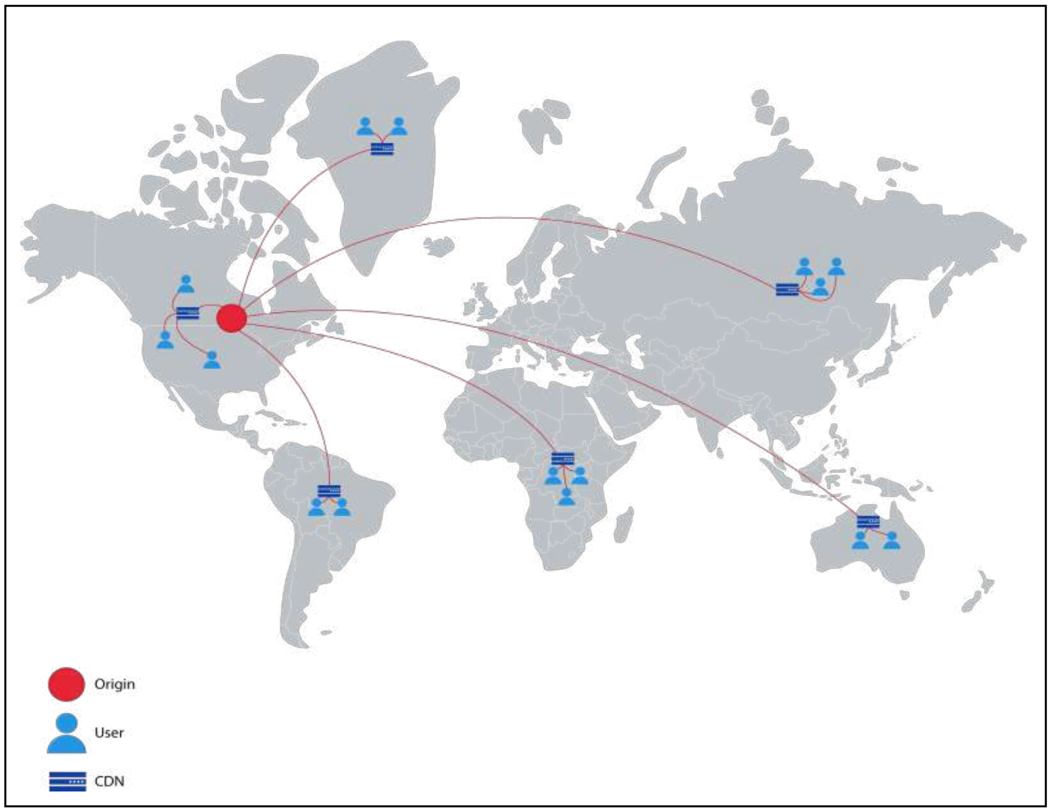
We can bring data close to the user by placing a small data center near the user and storing copies of the data there. CDN mainly stores two types of data: static and dynamic. A CDN primarily targets propagation delay by bringing the data closer to its users. CDN providers make the extra effort tc have sufficient bandwidth available through the path and bring data closer to the users (possibly within their ISP). They also try to reduce transmission and queuing delays because the ISP presumably has more bandwidth available within the autonomous system.

Let’s look at the different ways CDN solves the discussed problems:

- **High latency**: CDN brings the content closer to end users. Therefore, it reduces the physical distance and latency.
- **Data-intensive applications**: Since the path to the data includes only the ISP and the nearby CDN components, there’s no issue in serving a large number of users through a few CDN components in a specific area. As shown below, the origin data center will have to provide the data to local CDN components only once, whereas local CDN components can provide data to different users individually. No user will have to download their own copy of data from the origin servers.

> **Note**: Various streaming protocols are used to deliver dynamic content by the CDN providers. For example, CDNsun uses the Real-time Messaging Protocol (RTMP), HTTP Live Streaming (HLS), Real-time Streaming Protocol (RTSP), and many more to deliver dynamic content.

- **Scarcity of data center resources**: A CDN is used to serve popular content. Due to this reason, most of the traffic is handled at the CDN instead of the origin servers. So, different local or distributed CDN components share the load on origin servers.



Dissemination of content to a geographically distributed CDN

> **Note:** A few well-known CDN providers are Akamai, StackPath, Cloudflare, Rackspace, Amazon CloudFront, and Google Cloud CDN.

Point to Ponder

Question

Does a CDN cache all content from the origin server?

Show Answer

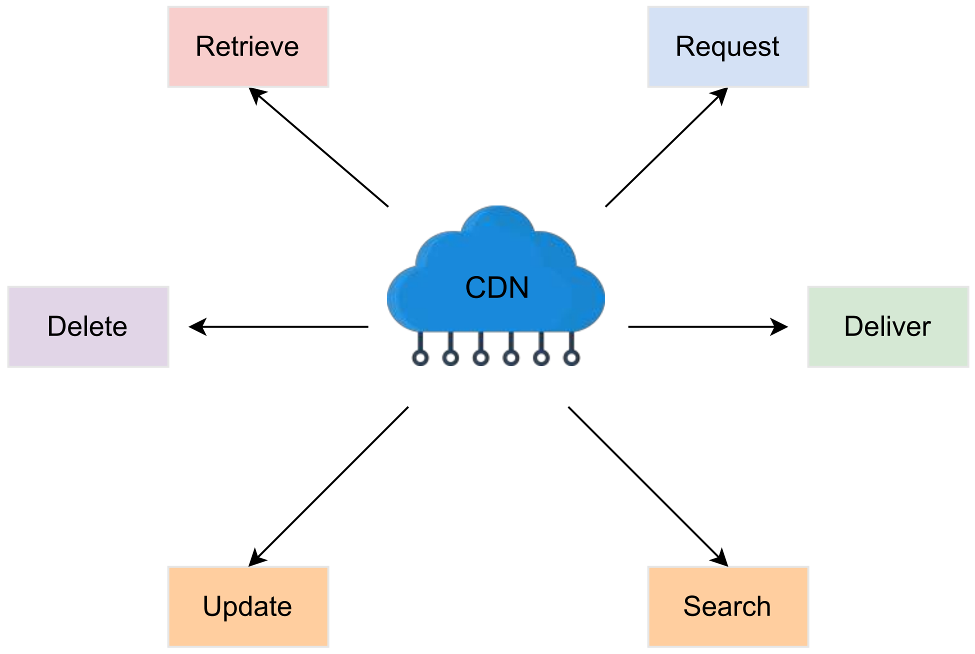
Requirements

Let's look at the functional and non-functional requirements that we expect from a CDN.

Functional requirements

The following functional requirements will be a part of our design:

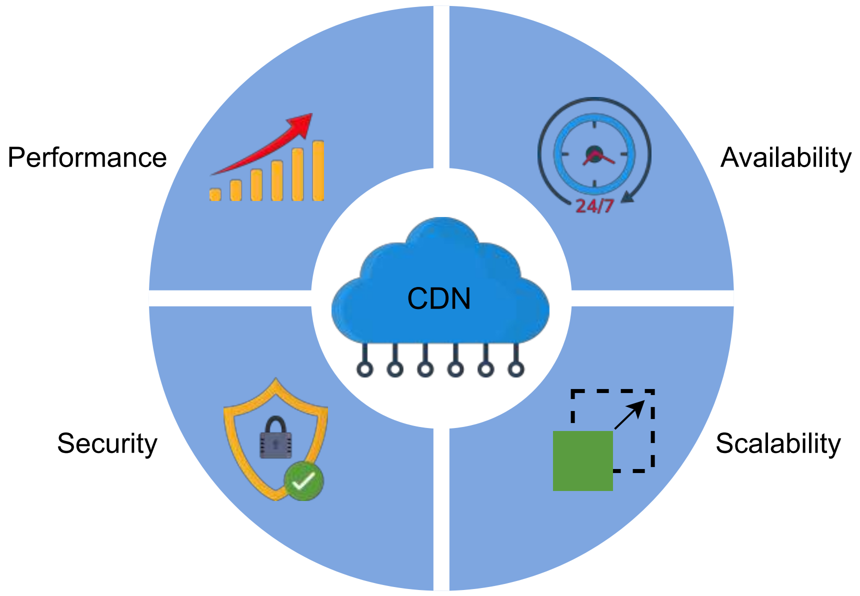
- **Retrieve**: Depending upon the type of CDN models, a CDN should be able to retrieve content from the origin servers. We'll cover CDN models in the coming lesson.
- **Request**: Content delivery from the proxy server is made upon the user's request. CDN proxy servers should be able to respond to each user's request in this regard.
- **Deliver**: In the case of the push model, the origin servers should be able to send the content to the CDN proxy servers.
- **Search**: The CDN should be able to execute a search against a user query for cached or otherwise stored content within the CDN infrastructure.
- **Update**: In most cases, content comes from the origin server, but if we run script in CDN, the CDN should be able to update the content within peer CDN proxy servers in a PoP.
- **Delete**: Depending upon the type of content (static or dynamic), it should be possible to delete cached entries from the CDN servers after a certain period.



Functional requirements of a CDN

Non-functional requirements

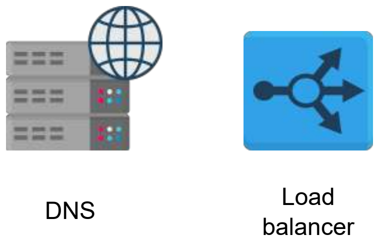
- **Performance**: Minimizing latency is one of the core missions of a CDN. The proposed design should have minimum possible latency.
- **Availability**: CDNs are expected to be available at all times because of their effectiveness. Availability includes protection against attacks like DDoS.
- **Scalability**: An increasing number of users will request content from CDNs. Our proposed CDN design should be able to scale horizontally as the requirements increase.
- **Reliability and security**: Our CDN design should ensure no single point of failure. Apart from failures, the designed CDN must reliably handle massive traffic loads. Furthermore, CDNs should provide protection to hosted content from various attacks.



Non-functional requirements of CDN

Building blocks we will use

The design of a CDN utilizes the following building blocks:



The building blocks used in CDN design

Design of a CDN

Let's understand the basic design of a CDN system.

We'll cover the following

CDN design

We'll explain our CDN design in two phases. In the first phase, we'll cover the components that comprise a CDN. By the end of this phase, we'll understand why we need a specific component. In the second phase, we'll explore the workflow by explaining how each component interacts with others to develop a fully functional CDN. Let's dive in.

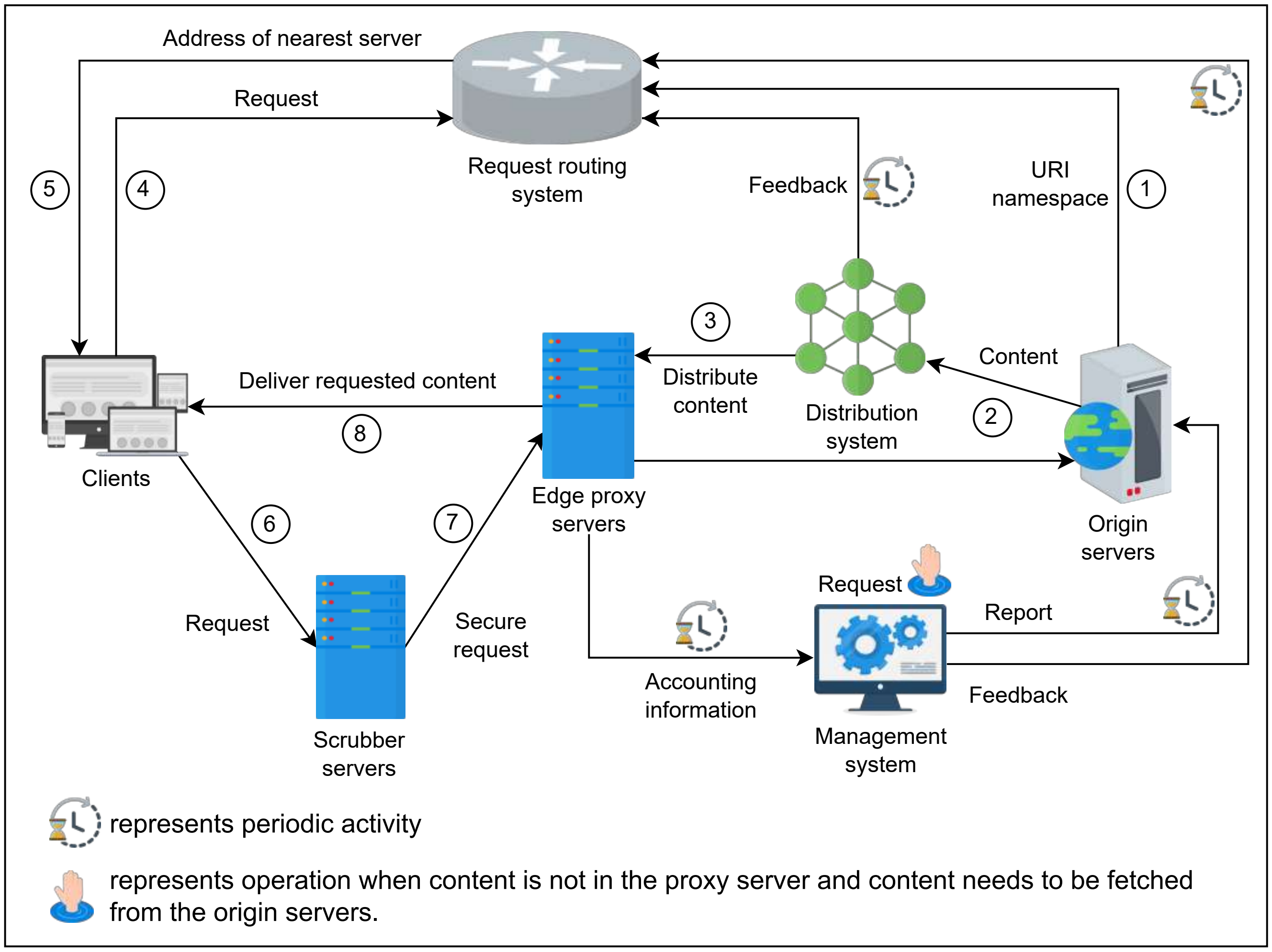
CDN components

The following components comprise a CDN:

- **Clients**: End users use various clients, like browsers, smartphones, and other devices, to request content from the CDN.
- **Routing system**: The routing system directs clients to the nearest CDN facility. To do that effectively, this component receives input from various systems to understand where content is placed, how many requests are made for particular content, the load a particular set of servers is handling, and the URI (Uniform Resource Identifier) namespace of various contents. In the next lesson, we'll discuss different routing mechanisms to forward users to the nearest CDN facility.
- **Scrubber servers**: Scrubber servers are used to separate the good

like DDoS. Scrubber servers are generally used only when an attack is detected. In that case, the traffic is scrubbed or cleaned and then routed to the target destination.
- **Proxy servers**: The proxy or edge proxy servers serve the content from RAM to the users. Proxy servers store hot data in RAM, though they can store cold data in SSD or hard drive as well. These servers also provide accounting information and receive content from the distribution system.
- **Distribution system**: The distribution system is responsible for distributing content to all the edge proxy servers to different CDN facilities. This system uses the Internet and intelligent broadcast-like approaches to distribute content across the active edge proxy servers.
- **Origin servers**: The CDN infrastructure facilitates users with data received from the origin servers. The origin servers serve any unavailable data at the CDN to clients. Origin servers will use appropriate stores to keep content and other mapping metadata. Though, we won't discuss the internal architecture of origin infrastructure here.
- **Management system**: The management systems are important in CD from a business and managerial aspect where resource usage and

statistics are constantly observed. This component measures important metrics, like latency, downtime, packet loss, server load, and so on. For third-party CDNs, accounting information can also be used for billing purposes.



CDN components

Workflow

The workflow for the abstract design is given below:

1. The origin servers provide the URI namespace delegation of all objects cached in the CDN to the request routing system.
2. The origin server publishes the content to the distribution system responsible for data distribution across the active edge proxy servers.

3. The distribution system distributes the content among the proxy servers and provides feedback to the request routing system. This feedback is helpful in optimizing the selection of the nearest proxy server for a requesting client. This feedback contains information about which content is cached on which proxy server to route traffic to relevant proxy servers.
4. The client requests the routing system for a suitable proxy server from the request routing system.
5. The request routing system returns the IP address of an appropriate proxy server.
6. The client request routes through the scrubber servers for security reasons.
7. The scrubber server forwards good traffic to the edge proxy server.
8. The edge proxy server serves the client request and periodically forwards accounting information to the management system. The management system updates the origin servers and sends feedback to the routing system about the statistics and detail of the content. However, the request is routed to the origin servers if the content isn't available in the proxy servers. It's also possible to have a hierarchy of proxy servers if the content isn't found in the edge proxy servers. For such cases, the request gets forwarded to the parent proxy servers.

API Design

This section will discuss the API design of the functionalities offered by CDN. This will help us understand how the CDN will receive requests from the clients, receive content from the origin servers, and communicate to other components in the network. Let's develop APIs for each of the following functionalities:

- Retrieve content
- Deliver content

- Request content
- Search content
- Update content
- Delete content

Content can be anything, like a file, video, audio, or other web object. Here, we'll use the word "content" to refer to all of the above. For clarity, we won't discuss the privacy-related parameters—like if the content is public or private, who should be able to access this content, if it should be encrypted, and so on—in the following APIs.

Retrieve (proxy server to origin server)

If the proxy servers request content, the `GET` method retrieves the content through the `/retrieveContent` API below:

```
retrieveContent(proxyserver_id, content_type, content_version, description)
```

Let's see the details of the parameters:

Details of Parameters

Parameter	Description
`proxyserver_id`	This is a unique ID of the requesting proxy server.
`content_type`	This data structure will contain information about the requested con Specifically, it will contain the category (audio, video, document, scri on), the type of clients it's requested for, and the requested quality
`content_version`	This represents the version number of the content. For the `/retrieve` the `content_version` will contain the current version of the conter the proxy server. The `content_version` will be `NULL` if no previous ve available at the proxy server.

`description`	This specifies the content detail—for example, the video's extension, detail, and so on if the `content_type` is video.

The above API gives a response in a JSON file, which contains the text, content types, links to the images or videos in the content, and so on.

Click to see the links in the JSON file from where various objects will be downloaded at the proxy servers.

Deliver (origin server to proxy servers)

The origin servers use this API to deliver the specified content, theupdated version, to the proxy servers through the distribution system. We call this the `/deliverContent` API:

```
deliverContent(origin_id, server_list, content_type, content_version, description)
```

Details of Parameters

Parameter	Description
`origin_id`	This recognizes each origin server uniquely.
`server_list`	This identifies the list of servers the content will be pushed to by the distribution system.
`content_version`	This represents the updated version of the content at the origin se proxy server receiving the content will discard the previous version.

The rest of the parameters have been explained above already.

Request (clients to proxy servers)

The users use this API to request the content from the proxy servers. We call this the `/requestContent` API:

```
requestContent(user_id, content_type, description)
```

Details of Parameter

Parameter	Description
`user_id`	This is the unique ID of the user who requested the content.

The specified proxy server returns the particular content to the requested users in response to the above API.

Click to see the links in the JSON file from where various objects will be downloaded at the user end.

Search (proxy server to peer proxy servers)

Although the content is first searched locally at the proxy server, the proxy servers can also probe requested content in the peer proxy servers in the same PoP through the `/searchContent` API. This could flood the query to all proxy servers in a PoP. Alternatively, we can use a data store in the PoP to query the content, though proxy servers will need to maintain what conte
is available on which proxy server.

The `/searchContent` API is shown below:

```
searchContent(proxyserver_id, content_type, description)
```

Update (proxy server to peer proxy servers)

The proxy servers use the `/updateContent` API to update the specified content in the peer proxy servers in the PoP. It does so when specified isolated scripts run on the CDN to provide image resizing, video resolution conversion, security, and many more services. This type of scripting is known as serverless scripting.

The `/updateContent` API is shown below:

```
updateContent(proxyserver_id, content_type, description)
```

Details of Parameter

Parameter	Description
`porxyserver_id`	This recognizes the proxy server uniquely in the PoP to update the content.

The rest of the parameters have been explained above already.

> **Note**: The Delete API isn't discussed here. In our caching chapter, we discussed different eviction mechanisms in detail. Those mechanisms are also applicable for a CDN content eviction. Nevertheless, situations can arise where the Delete APIs may be required. We'll discuss a few content consistency mechanisms, like how much time content stays in the cache, in the next lesson.

In the upcoming lessons, we'll dive deep into the characteristics of CDNs.

← Back

Introduction to a CDN

Next →

In-depth Investigation of CDN: Part 1

✓ Mark as Completed

In-depth Investigation of CDN: Part 1

Learn push and pull models and dynamic content cache optimization in CDNs.

We'll cover the following

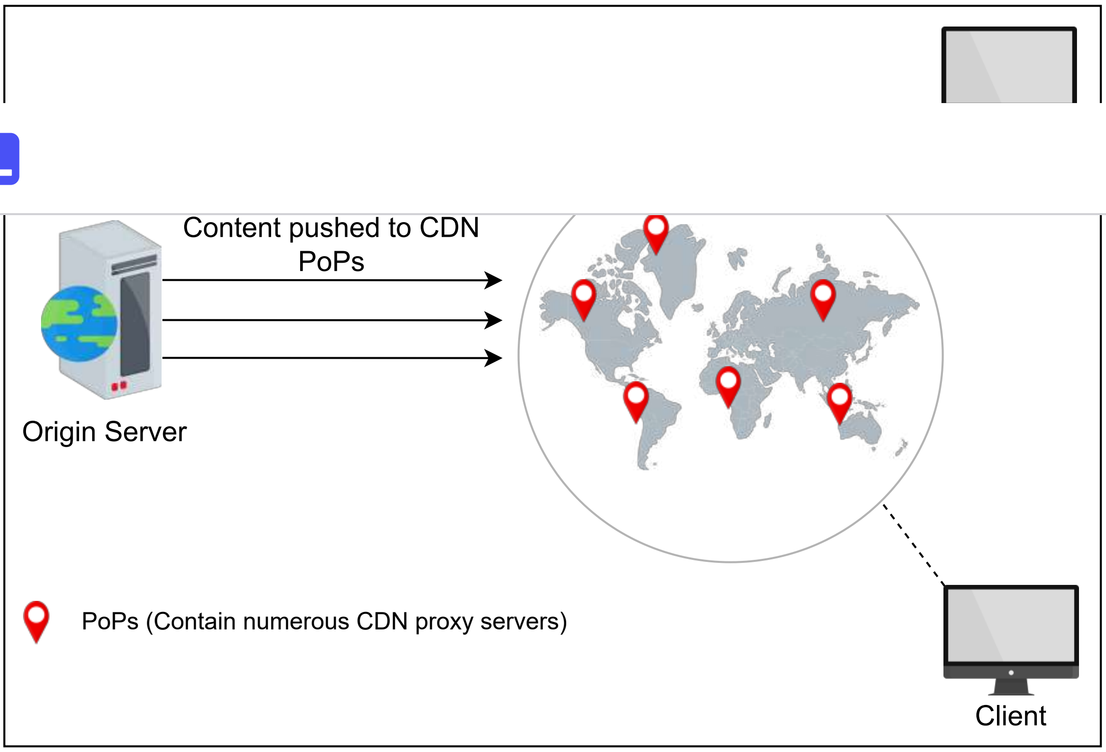
In this lesson, we'll go into the details of certain concepts, such as CDN models and multi-tier/layered CDN architecture, that we mentioned in the previous lessons. We'll also introduce some new concepts, including dynamic content caching optimization and various techniques to discover the nearby proxy servers in CDNs.

Content caching strategies in CDN

Identifying content to cache is important in delivering up-to-date and popular web content. To ensure timely updates, two classifications of CDNs are used to get the content from the origin servers.

Push CDN

Content gets sent automatically to the CDN proxy servers from the origin server in the push CDN model. The content delivery to the CDN proxy servers is the content provider's responsibility. Push CDN is appropriate for static content delivery, where the origin server decides which content to deliver to users using the CDN. The content is pushed to proxy servers in various locations according to the content's popularity. If the content is rapidly changing, the push model might struggle to keep up and will do redundant content pushes.

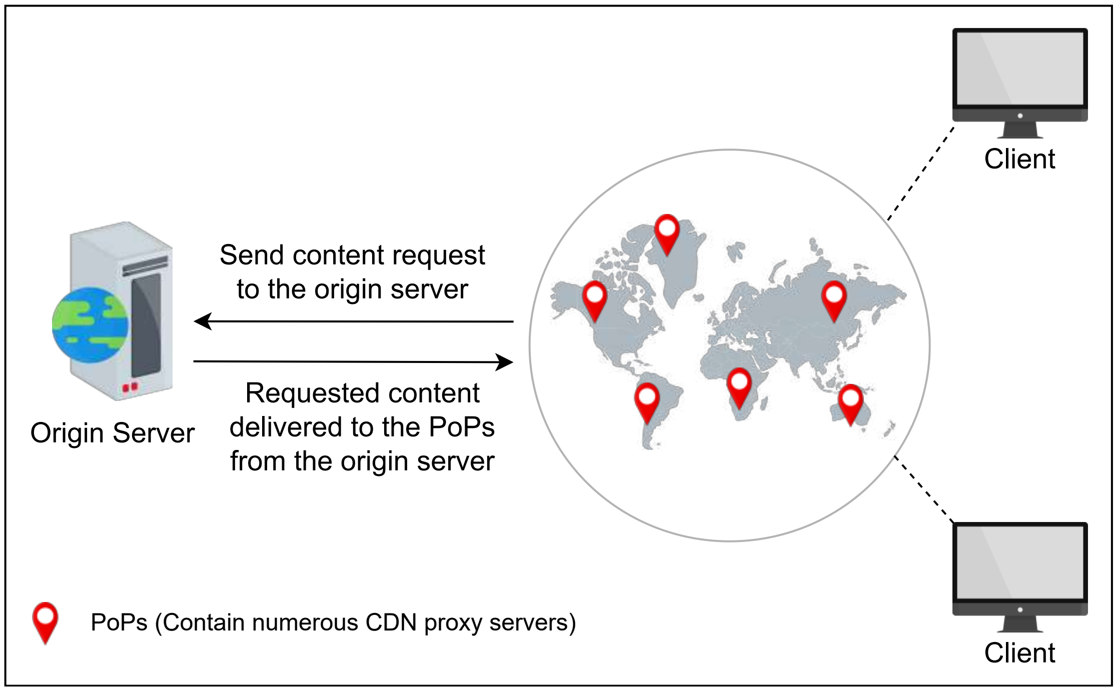

Content pushed to CDN PoPs
Origin Server
PoPs (Contain numerous CDN proxy servers)
Client

Push content to PoPs

Pull CDN

A CDN pulls the unavailable data from origin servers when requested by a user. The proxy servers keep the files for a specified amount of time and then remove them from the cache if they're no longer requested to balance capacity and cost.

When users request web content in the pull CDN model, the CDN itself is responsible for pulling the requested content from the origin server and serving it to the users. Therefore, this type of CDN is more suited for serving dynamic content.


Client
Send content request to the origin server
Origin Server
Requested content delivered to the PoPs from the origin server
PoPs (Contain numerous CDN proxy servers)
Client

Content pull from origin server to the CDN PoPs

As stated, the push CDN is mostly used for serving static content. Since static content is served to a wide range of users for longer than dynamic content, the push CDN scheme maintains more replicas than the pull CDN, thus improving availability. On the other hand, the pull CDN is favored for frequently changing content and a high traffic load. Low storage consumption is one of the main benefits of the pull CDN.

> **Note:** Most content providers use both pull and push CDN caching approaches to get the benefits of both.

Dynamic content caching optimization

Since dynamic content often changes, it’s a good idea to cache it optimally. This section deals with the optimization of frequently changing content.

Certain dynamic content creation requires the execution of scripts that can be executed at proxy servers instead of running on the origin server. Dynamic data can be generated using various parameters, which can be beneficial if executed at the proxy servers. For example, we can generate dynamic content based on user location, time of day at a location, third-party APIs specific to a location (for instance, weather API), and so on. So, it’s optimal to run the scripts at proxy servers instead of the origin servers.

To reduce the communication between the origin server and proxy servers and storage requirements at proxy servers, it’s useful to employ compression techniques as well. For example, Cloudflare uses Railgun to compress dynamic content.

Another popular approach for dynamic data compression is **Edge Side Includes (ESI)** markup language. Usually, a small portion of the web pages changes in a certain time. It means fetching a full web page on each small change contains a lot of redundant data. To resolve this performance penalty, ESI specifies where content was changed so that the rest of the web page content can be cached. It assembles dynamic content at the CDN edge server or client browser. ESI isn’t standardized yet by the World Wide Web Consortium (W3C), but many CDN providers use it.

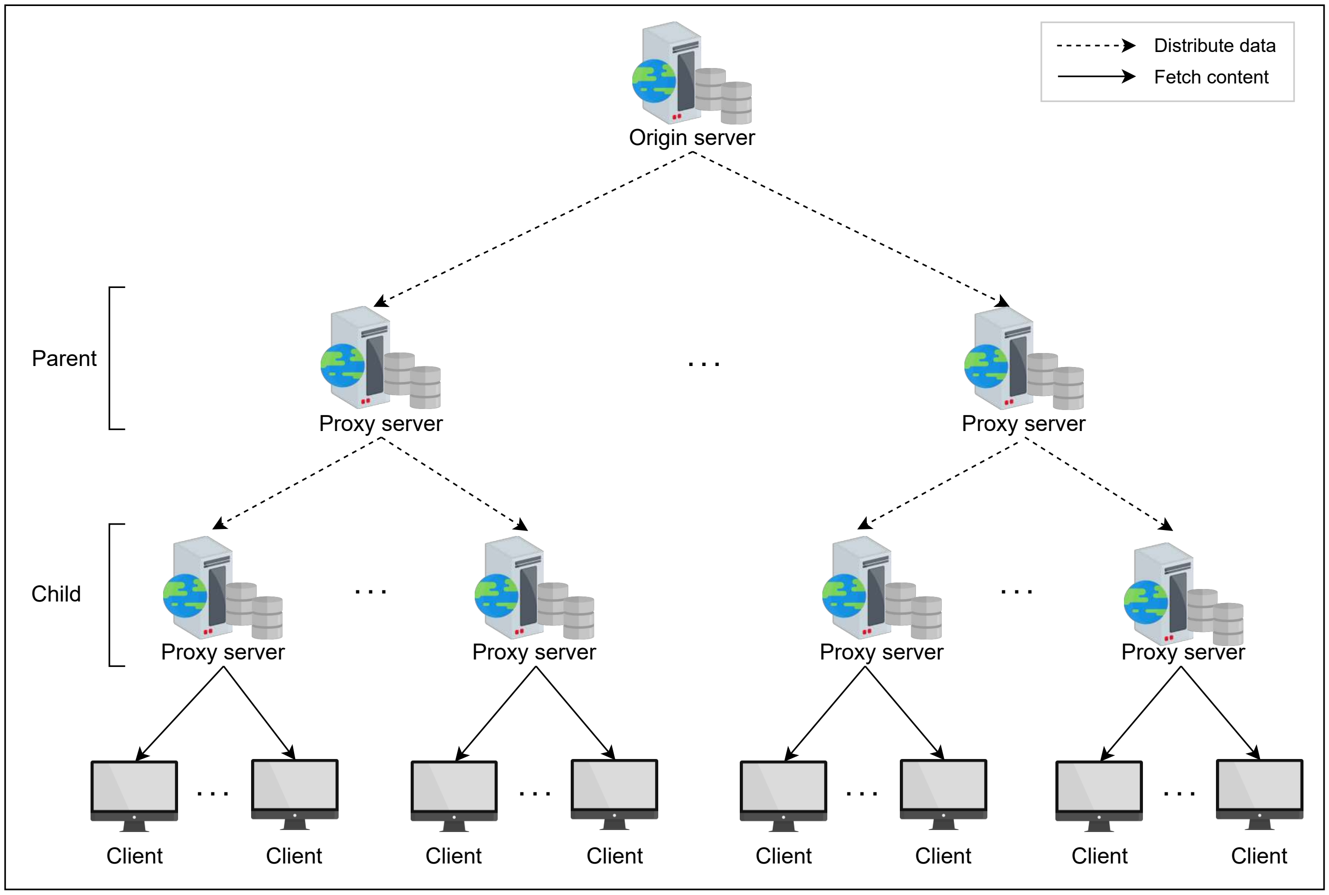
> **Note**: **Dynamic Adaptive Streaming one HTTP (DASH)** uses a manifest file with URIs of the video with different resolutions so that the client can fetch whatever is appropriate as per prevailing network and end node conditions. Netflix uses a proprietary DASH

> version with a Byte-range in the URL for further content request and delivery optimization.

Multi-tier CDN architecture

The content provider sends the content to a large number of clients through a CDN. The task of distributing data to all the CDN proxy servers simultaneously is challenging and burdens the origin server significantly. CDNs follow a tree-like structure to ease the data distribution process for the origin server. The edge proxy servers have some peer servers that belong to the same hierarchy. This set of servers receives data from the parent nodes in the tree, which eventually receive data from the origin servers. The data is copied from the origin server to the proxy servers by following different paths in the tree.

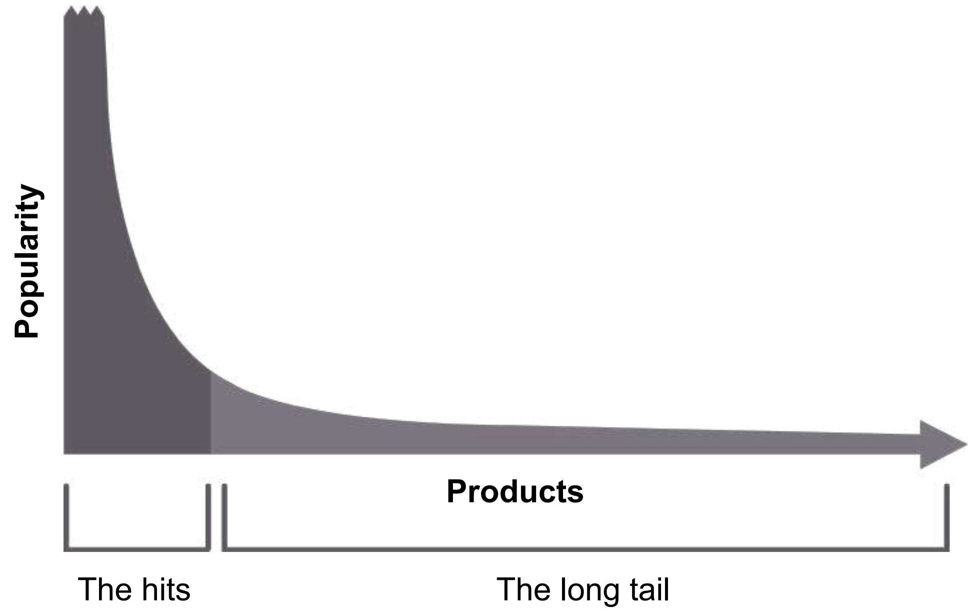
The tree structure for data distribution allows us to scale our system for increasing users by adding more server nodes to the tree. It also reduces the burden on the origin server for data distribution. A CDN typically has one or two tiers of proxy servers (caches). The following illustration shows the two tiers of proxy servers:



Data distribution among CDN proxy servers

Whenever a new proxy server enters the tree of a CDN, it requests the control core, which maintains information on all the proxy servers in the CDN and provides initial content with the configuration data.

Research shows that many contents have long-tail distribution. This means that, at some point, only a handful of content is very popular, and then we have a long tail of less popular content. Here, a **multi-layer cache** might be used to handle long-tail content.



Many kinds of data exhibit the long-tailed phenomenon

Point to Ponder

Question

What happens if a child or parent proxy server fails or if the origin server fails?

Show Answer

Now that we've seen a way to distribute content from the origin server to all the proxy servers of the CDN, we should also educate ourselves on how use can use these proxy servers to get the data more efficiently. We'll discuss how the nearest proxy server is chosen when clients make requests and hc the CDN is located in the upcoming sections of this lesson.

Find the nearest proxy server to fetch the data

It’s vital for the user to fetch data from the nearest proxy server because the CDN aims to reduce user-perceived latency by bringing the data close to the user. However, the question remains of how users worldwide request data from the nearest proxy server. The goal of this section is to answer that question.

Important factors that affect the proximity of the proxy server

There are two important factors that are relevant to finding the nearest proxy server to the user:

- **Network distance** between the user and the proxy server is crucial. This is a function of the following two things:
  - The first is the length of the network path.
  - The second is the capacity (bandwidth) limits along the network path.

The shortest network path with the highest capacity (bandwidth) is the nearest proxy server to the user in question. This path helps the user download content more quickly.

- **Requests load** refers to the load a proxy server handles at any point in time. If a set of proxy servers are overloaded, the request routing system should forward the request to a location with a lesser load. This action balances out the proxy server load and, consequently, reduces the response latency.

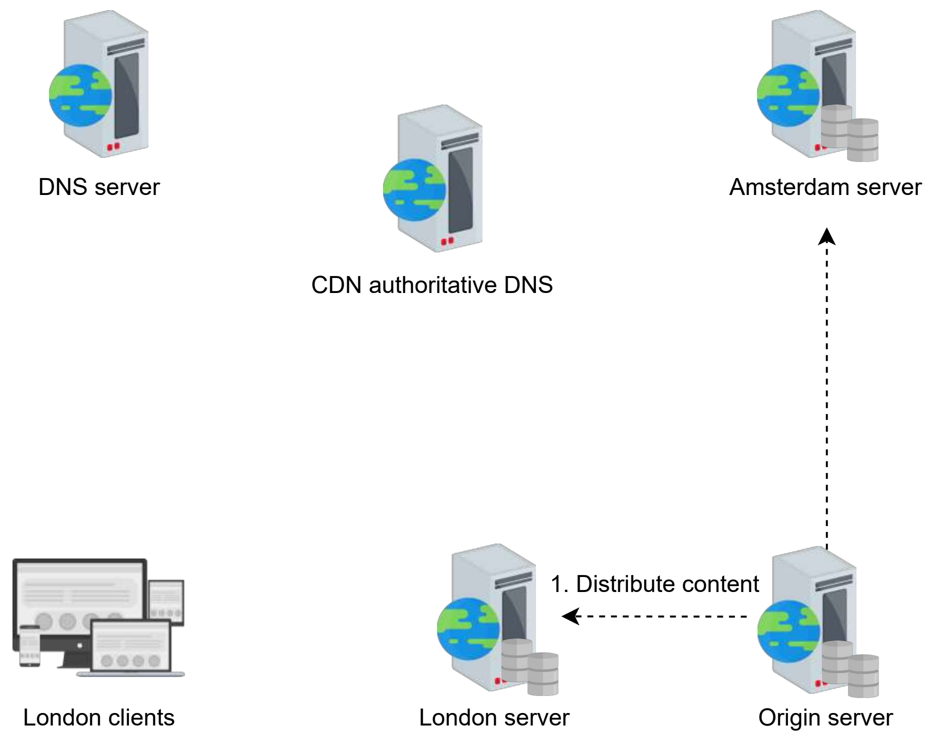
Let’s look at the techniques that can be used to route users to the nearest proxy server.

DNS redirection

In a typical *DNS resolution*, we use a DNS system to get an IP against a human-readable name. However, the DNS can also return another URI (instead of an IP) to the client. Such a mechanism is called **DNS redirect**.

Content providers can use DNS redirect to send a client to a specific CDN. As an example, if the client tries to resolve a name that has the word "video" in it, the authoritative DNS server provides another URL (for example, *cdn.xyz.com*). The client does another DNS resolution, and the CDN's authoritative DNS provides an IP address of an appropriate CDN proxy server to fetch the required content.

Depending on the location of the user, the response of the DNS can be different. Let's see the slides below to understand how DNS redirection works:



The origin server distributes content to the proxy servers

1 of

> **Note**: The nearest proxy server doesn't necessarily mean the one that's geographically the closest. It could be, but it's not only the geography that matters. Other factors like network distance, bandwidth, and traffic load already on that route also matter.

There are two steps in the DNS redirection approach:

1. In the first step, it maps the clients to the appropriate network location.
2. In the second step, it distributes the load over the proxy servers in that location to balance the load among the proxy servers (see DNS and Load Balancers building blocks for more details on this).

DNS redirection takes both of these important factors—network distance and requests load—into consideration, and that reduces the latency towards a proxy server.

To shift a client from one machine in a cluster to another, the DNS replies at the second step are given with short TTLs so that the client repeats the resolution after a short while. DNS keeps delivering the content by routing requests to other active servers in case of hardware failures and network congestion. It does so by load balancing the traffic, using intelligent failover, and maintaining servers across many data centers, which achieves good reliability and performance.

Since the load at proxy servers changes over time, the content provider needs to make appropriate changes in the DNS to make the DNS redirection effective. Many CDN providers like Akamai use DNS redirection in their routing system.

Anycast

Anycast is a routing methodology in which all the edge servers located in multiple locations share the same single IP address. It employs the **Border Gateway Protocol (BGP)** to route clients based on the Internet’s natural network flow. A CDN provider can use the anycast mechanism so that clients are directed to the nearest proxy servers for content.

Client multiplexing

Client multiplexing involves sending a client a list of candidate servers. The client then chooses one server from the list to send the request to. This approach is inefficient because the client lacks the overall information to choose the most suitable server for their request. This may result in sending requests to an already-loaded server and experiencing higher access latency.

HTTP redirection

HTTP redirection is the simplest of all approaches. With this scheme, the client requests content from the origin server. The origin server responds with an HTTP protocol to redirect the user via a URL of the content.

Below is an example of an HTML snippet provided by Facebook. As is highlighted in line 8, the user is redirected to the CDN to download the logo of Facebook:

```
<!--  The code below is taken from Facebook. -->
<div class="fb_content clearfix " id="content" role="main">
 <div>
```

In-depth Investigation of CDN: Part 2

Learn about content consistency mechanisms and the deployment of the proxy server in a CDN.

We'll cover the following

In this lesson, we learn how content consistency can be achieved using different consistency mechanisms. We also learn about where we should deploy the proxy servers and the difference between CDN as a service and specialized CDN.

Content consistency in CDN

Data in the proxy servers should be consistent with data in the origin servers. There's always a risk of users accessing stale data if the proxy servers don't remain consistent with the origin servers. Different consistency mechanisms can be used to ensure consistency of data, depending on the push or pull model.

Periodic polling

Using the pull model, proxy servers request the origin server periodically for updated data and change the content in the cache accordingly. When content changes infrequently, the polling approach consumes unnecessary bandwidth. Periodic polling uses **time-to-refresh (TTR)** to adjust the time period for requesting updated data from the origin servers.

Time-to-live (TTL)

Because of the TTR, the proxy servers may uselessly request the origin servers for updated data. A better approach that could be employed to reduce the frequency of refresh messages is the **time-to-live (TTL)** approach. In this approach, each object has a TTL attribute assigned to it by the origin server. The TTL defines the expiration time of the content. The proxy servers serve the same data version to the users until that content expires. Upon expiration, the proxy server checks for an update with the origin server. If the data is changed, it gets the updated data from the origin server and then responds to the user's requests with the updated data. Otherwise, it keeps the same data with an updated expiration time from the origin servers.

Leases

The origin server grants a lease to the data sent to a proxy server using this technique. The **lease** denotes the time interval for which the origin server agrees to notify the proxy server if there's any change in the data. The proxy server must send a message requesting a lease renewal after the expiratio of the lease. The lease method helps to reduce the number of messages exchanged between the proxy and origin server. Additionally, the lease duration can be optimized dynamically according to the observed load on the proxy servers. This technique is referred to as an **adaptive lease**.

In the following section, we discuss where to place the CDN proxy server to transmit data effectively.

We have to be clear with the answers to the following questions before we instal the CDN facility:

- What are the best locations to install proxy servers to maximally utilize CDN technology?
- How many CDN proxy servers should we install?

Placement of CDN proxy servers

The CDN proxy servers must be placed at network locations with good connectivity. See the options below:

- **On-premises** represents a smaller data center that could be placed near major IXPs.
- **Off-premises** represents placing CDN proxy servers in ISP's networks.

Today, it might be feasible to keep a large portion of a movie's data in a CDN infrastructure that's housed inside an ISP. Still, for services like YouTube, data is so large and ever-expanding that it's challenging to decide what we should put near a user. Google uses split TCP to reduce user-perceived delays by keeping persistent connections with huge TCP windows from the IXP-level infrastructure to their primary data centers. The client's TCP requests terminate at the IXP-level infrastructure and are then forwarded on already established, low latency TCP connections.

Doing this substantially reduces client-perceived latency, which is due to tl avoidance of the initial three-way handshake of TCP connection and slow-start stages to a host far away (had the client wanted to go to the primary data centers of Google). A round-trip delay to IXP is often very low. Therefore, three-way handshakes and slow starts at that level are negligible. **Predictive push** is a significant research field to decide what to push near the customers.

We can use measurements to facilitate the decision of proxy server placement. One such tool is **ProxyTeller** to decide where to place the proxy server and how many proxy servers are required to achieve high performance. ProxyTeller uses hit ratio, network bandwidth, and client-response time (latency) as performance parameters to decide the placement of proxy servers. Other greedy, random, and hotspot algorithms are also used for proxy server placements.

> **Note**: Akamai and Netflix popularized the idea of keeping their CDN proxy servers inside the client's ISPs. For many clients of Akamai, content is just one network hop away. On the other hand, Google also has its private CDN infrastructure but relies more on its servers near IXPs. One reason for this could be the sheer amount of data and the change patterns.

Point to Ponder

Question

What benefits could an ISP get by placing the CDN proxy servers inside their network?

Show Answer

CDN as a service

Most companies don't build their own CDN. Instead, they use the services of a CDN provider, such as Akamai, Cloudflare, Fastly, and so on, to deliver their content. Similarly, players like AWS make it possible for anyone to use a global CDN facility.

The companies sign a contract with the CDN service provider and deliver their content to the CDN, thereby allowing the CDN to distribute the content to the end users. A public CDN raises the following concerns for content providers:

- The content provider can't do anything if the public CDN is down.
- If a public CDN doesn't have any proxy servers located in the region or country where some website traffic comes from, then those specific customers are out of luck. In such cases, the content providers have to buy CDN services from other CDN providers or deploy and use their own private CDN.
- It's possible that some domains or IP addresses of CDN providers are blocked or restricted in some countries because they might be delivering content that's banned in those countries.

> **Note**: Some companies make their own CDN instead of using the services of CDN providers. For example, Netflix has its own purpose-built CDN called **Open Connect**.

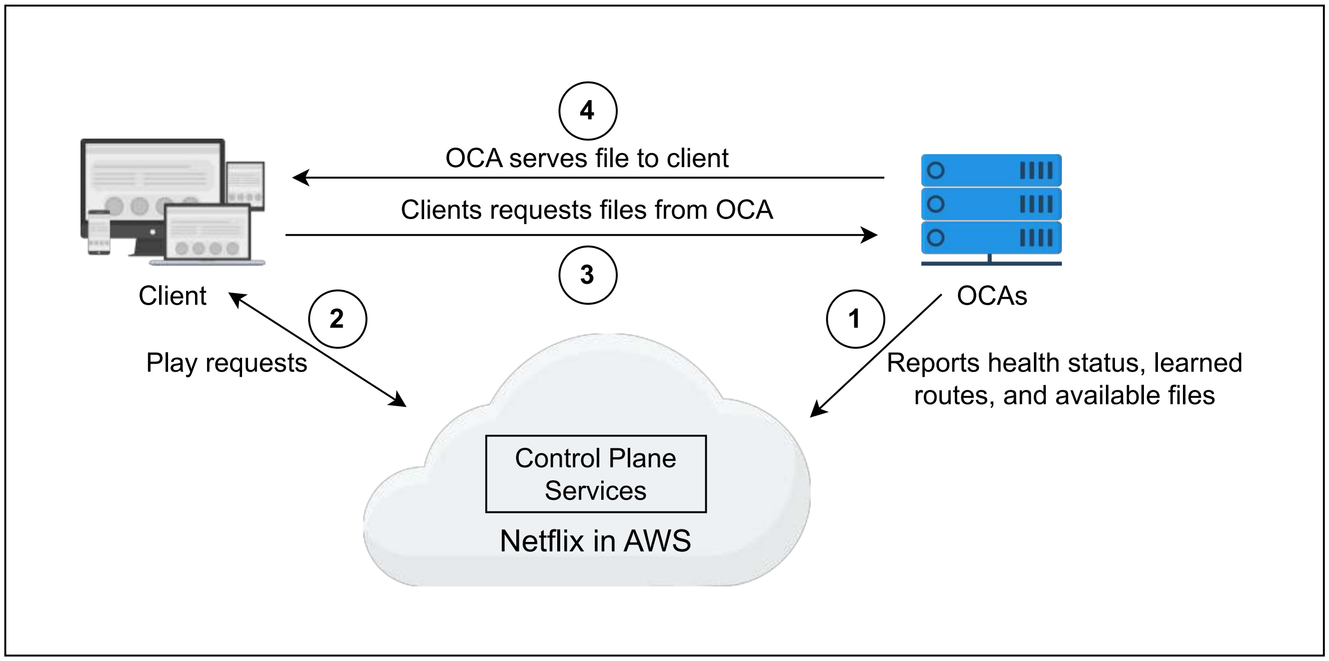
Specialized CDN

We’ve discussed that many companies use CDN as a service, but there are cases where companies build their own CDN. A number of reasons factor into this decision. One is the cost of a commercial CDN service. A specialized CDN consists of points of presence **(PoPs)** that only serve content for their own company. These PoPs can be caching servers, reverse proxies, or application delivery controllers. Although a specialized CDN has high costs at its first setup, the costs eventually decrease with time. In essence, it’s a buy versus build decision.

The specialized CDN’s PoPs consist of many proxy servers to serve petabytes of content. A private CDN can be used in coexistence with a public CDN. In case the capacity of a private CDN isn’t enough or there’s a failure that leads to capacity reduction, the public CDN is used as a backup. Netflix’s **Open Connect Appliance (OCA)** is an example of a CDN that’s specialized in video delivery.

Netflix’s OCA servers don’t store user data. Instead, they fulfill the following tasks:

- They report their status—health, learned routes, and details of cached content—to the Open Connect control plane that resides in AWS (Amazon Web Services).
- They serve the cached content that’s requested by the user.


4
OCA serves file to client
Clients requests files from OCA
3
Client
OCAs
2
Play requests
1
Reports health status, learned routes, and available files
Control Plane Services
Netflix in AWS

Netflix's Open Connect Appliances

All the deployed OCAs situated in IXP or embedded in the ISP network are monitored by the Open Connect operation team.

Why Netflix built its CDN

As Netflix became more popular, it decided to build and manage its own CDN for the following reasons:

- The CDN service providers were scuffling to expand their infrastructure due to the rapid growth in customer demand for video streaming on Netflix.
- With the increasing volume of streaming videos, the expense of using CDN services increased.
- Video streaming is the main business and a primary revenue source for Netflix. So, protecting the data of all the videos on the platform is critical. Netflix's OCA manages potential data leakage risks in a better way.
- To provide optimal streaming media delivery to customers, Netflix needed to maximize its control over the user's video player, the

network between the user, and the Netflix servers.

- Netflix’s OCA can use custom HTTP modules and TCP connection algorithms to detect network problems quickly and troubleshoot any issues in their CDN network.
- Netflix wanted to keep popular content for a long time. This wasn’t entirely possible while operating with a public CDN due to the high costs that would be incurred to keep and maintain it.

> **Note**: Netflix is able to achieve a hit ratio close to 95% using OCA.

We’ll evaluate our proposed design in the next lesson

Design of a Unique ID Generator

Learn how to design a system that generates a unique ID.

We'll cover the following

In the previous lesson, we saw that we need unique identifiers for many use cases, such as identifying objects (for example, Tweets, uploaded videos, and so on) and tracing the execution flow in a complex web of services. Now, we'll formalize the requirements for a unique identifier and discuss three progressively improving designs to meet our requirements.

Requirements for unique identifiers

The requirements for our system are as follows:

- **Uniqueness**: We need to assign unique identifiers to different events for identification purposes.

- **Scalability**: The ID generation system should generate at least a billion unique IDs per day.
- **Availability**: Since multiple events happen even at the level of nanoseconds, our system should generate IDs for all the events that occur.
- **64-bit numeric ID**: We restrict the length to 64 bits because this bit size is enough for many years in the future. Let's calculate the number of years after which our ID range will wrap around.

  Total numbers available = 2^{64} = 1.8446744 x 10^{19}

  Estimated number of events per day = 1 billion = 10^9

  Number of events per year = 365 billion = 365×10^9

  Number of years to deplete identifier range = $\frac{2^{64}}{365 \times 10^9}$ = 50,539,024.8595 years

  64 bits should be enough for a unique ID length considering these calculations.

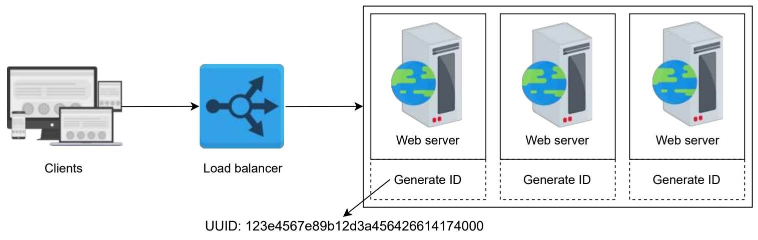
Let's dive into the possible solutions for the problem mentioned above.

First solution: UUID

A straw man solution for our design uses **UUIDs (universally unique IDs)**. This is a 128-bit number and it looks like $123e4567e89b12d3a456426614174000$ in hexadecimal. It gives us about 10^{38} numbers.

UUIDs have different versions. We opt for version 4, which generates a pseudorandom number.

Each server can generate its own ID and assign the ID to its respective event. No coordination is needed for UUID since it's independent of the server. Scaling up and down is easy with UUID, and this system is also highly available. Furthermore, it has a low probability of collisions. The design for this approach is given below:



Generating a unique ID using the UUID approach

Point to Ponder

Q

(Select all that apply.) What are the pros of using the UUID approach?

☐ A) It doesn't require synchronization between servers.

☐ B) It's a simple approach.

☐ C) It's scalable.

D) It's available.

Submit Answer

Cons

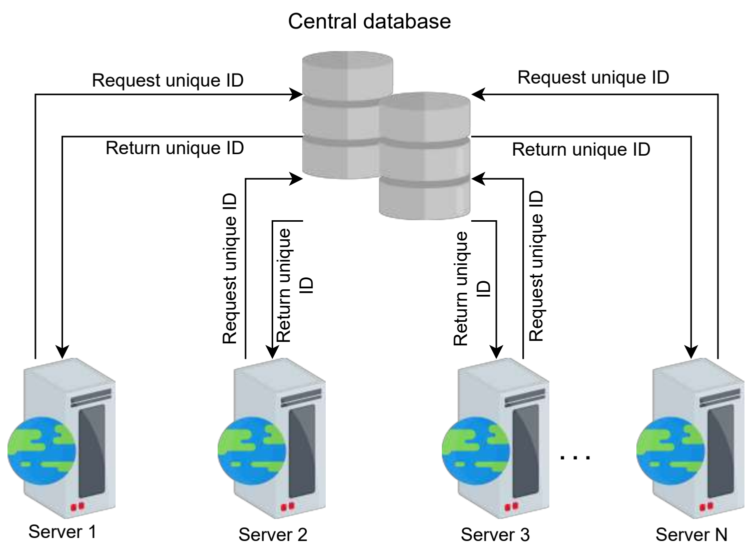
Using 128-bit numbers as primary keys makes the primary-key indexing slower, which results in slow inserts. A workaround might be to interpret an ID as a hex string instead of a number. However, non-numeric identifiers might not be suitable for many use cases. The ID isn't of 64-bit size. Moreover, there's a chance of duplication. Although this chance is minimal, we can't claim UUID to be deterministically unique. Additionally, UUIDs given to clients over time might not be monotonically increasing. The following table summarizes the requirements we have fulfilled using UUID:

Requirements Filled with UUID

	Unique	Scalable	Available
Using UUID	✗	✓	✓

Second solution: using a database

Let's try mimicking the auto-increment feature c a database. Consider a central database that provides a current ID and then increments the value by one. We can use the current ID as a unique identifier for our events.



Using a central database to generate unique IDs

Point to Ponder

Question

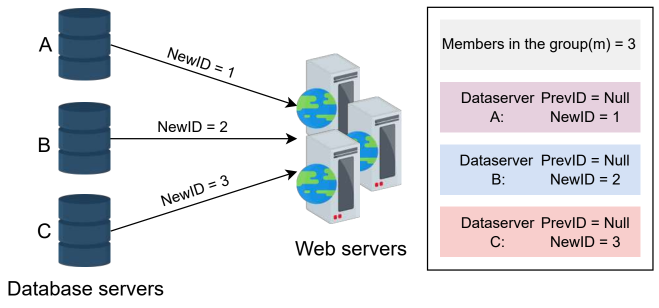
What can be a potential problem of using a central database?

Show Answer

To cater to the problem of a single point of failure. we modify the conventional auto-increment feature that increments by one. Instead of incrementing by one, let's rely on a value `m`, wher

`m` equals the number of database servers we hav

Each server generates an ID, and the following ID adds `m` to the previous value. This method is scalable and prevents the duplication of IDs. The following image provides a visualization of how a unique ID is generated using a database:


A
B
C
Database servers
NewID = 1
NewID = 2
NewID = 3
Web servers
Members in the group(m) = 3
Dataserver A: PrevID = Null NewID = 1
Dataserver B: PrevID = Null NewID = 2
Dataserver C: PrevID = Null NewID = 3

Generating IDs using the value of m

1 of 2

Pros

This approach is scalable. We can add more servers, and the value of `m` will be updated accordingly.

Cons

Though this method is somewhat scalable, it’s difficult to scale for multiple data centers. The task of adding and removing a server can result in duplicate IDs. For example, suppose `m=3`, and server A generates the unique IDs 1, 4, and 7. Server B generates the IDs 2, 5, and 8, while serv C generates the IDs 3, 6, and 9. Server B faces downtime due to some failure. Now, the value `m` is updated to 2. Server A generates 9 as its following unique ID, but this ID has already been generated by server C. Therefore, the IDs aren't unique anymore.

The table below highlights the limitations of our solution. A unique ID generation system shouldn't be a **single point of failure (SPOF)**. It should be scalable and available.

Requirements Filled by UUID versus Using a Database

	Unique	Scalable	Available
Using UUID	✗	✓	✓
Using a database	✗	✗	✓

Third solution: using a range handler

Let's try to overcome the problems identified in the previous methods. We can use ranges in a central server. Suppose we have multiple ranges for one to two billion, such as 1 to 1,000,000; 1,000,001 to 2,000,000; and so on. In such a case, a central microservice can provide a range to a server upon request.

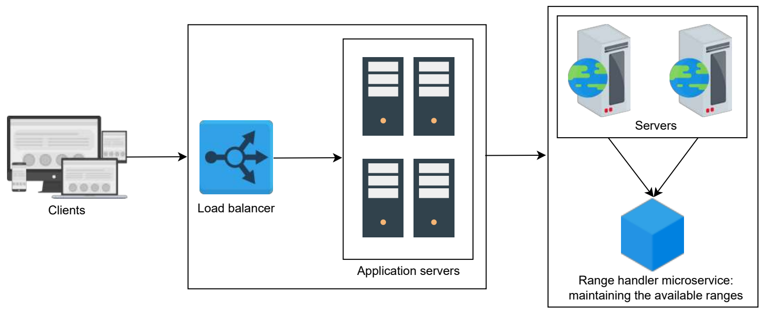
Any server can claim a range when it needs it fo the first time or if it runs out of the range. Suppos a server has a range, and now it keeps the start o the range in a local variable. Whenever a reques for an ID is made, it provides the local variable value to the requestor and increments the value by one.

Let’s say server 1 claims the number range 300,001 to 400,000. After this range claim, the user ID 300,001 is assigned to the first request. The server then returns 300,002 to the next user, incrementing its current position within the range. This continues until user ID 400,000 is released by the server. The application server then queries the central server for the next available range and repeats this process.

This resolves the problem of the duplication of user IDs. Each application server can respond to requests concurrently. We can add a load balancer over a set of servers to mitigate the load of requests.

We use a microservice called **range handler** that keeps a record of all the taken and available ranges. The status of each range can determine if a range is available or not. The state—that is, which server has what range assigned to it—can be saved on a replicated storage.

This microservice can become a single point of failure, but a **failover server** acts as the savior in that case. The failover server hands out ranges when the main server is down. We can recover the state of available and unavailable ranges from the latest checkpoint of the replicated store.


Clients
Load balancer
Application servers
Servers
Range handler microservice: maintaining the available ranges

Design of the range handler microservice

Pros

This system is scalable, available, and yields user IDs that have no duplicates. Moreover, we can maintain this range in 64 bits, which is numeric.

Cons

We lose a significant range when a server dies and can only provide a new range once it's live again. We can overcome this shortcoming by allocating shorter ranges to the servers, although ranges should be large enough to serve identifiers for a while.

The following table sums up what this approach fulfills for us:

Requirements Filled by These Three Options

	Unique	Scalable	Availa
Using UUID	✗	✓	
Using a database	✗	✗	

Using a range handler	✓	✓	✓

We developed a solution that provides us with a unique ID, which we can assign to various events and even use as a primary key. But what if we add a requirement that the ID is time sortable too?

Introduction to Distributed Monitoring

Learn why monitoring in a distributed system is crucial.

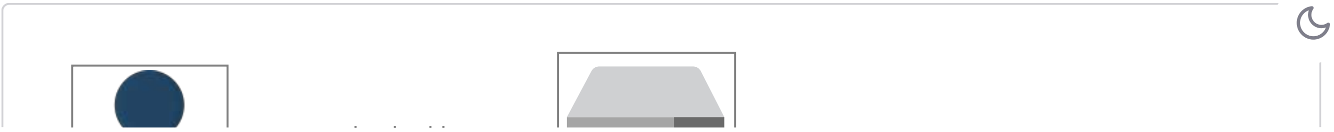
We'll cover the following

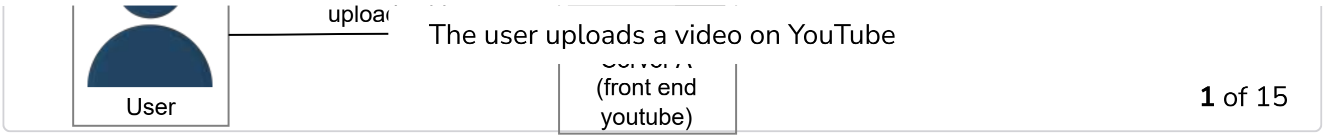
Need for monitoring

Let's go over how the failure of a single service can affect the smooth execution of related systems. To avoid cascading failures, monitoring can play a vital role with early warnings or steering us to the root cause of faults.

Let's consider a scenario where a user uploads a video, `intro-to-system-design`, to YouTube. The `UI` service in server `A` takes the video information and gives the data to service 2 in server B. Service 2 makes an entry in the database and stores the video in blob storage. Another service, 3, in server C manages the replication and synchronization of databases X and Y.

In this scenario, service 3 fails due to some error, and service 2 makes an entry in the database X. The database X crashes, and the request to fetch a video is routed to database Y. The user wants to play the video `intro-to-system-design`, but it will give an error of "Video not found..."



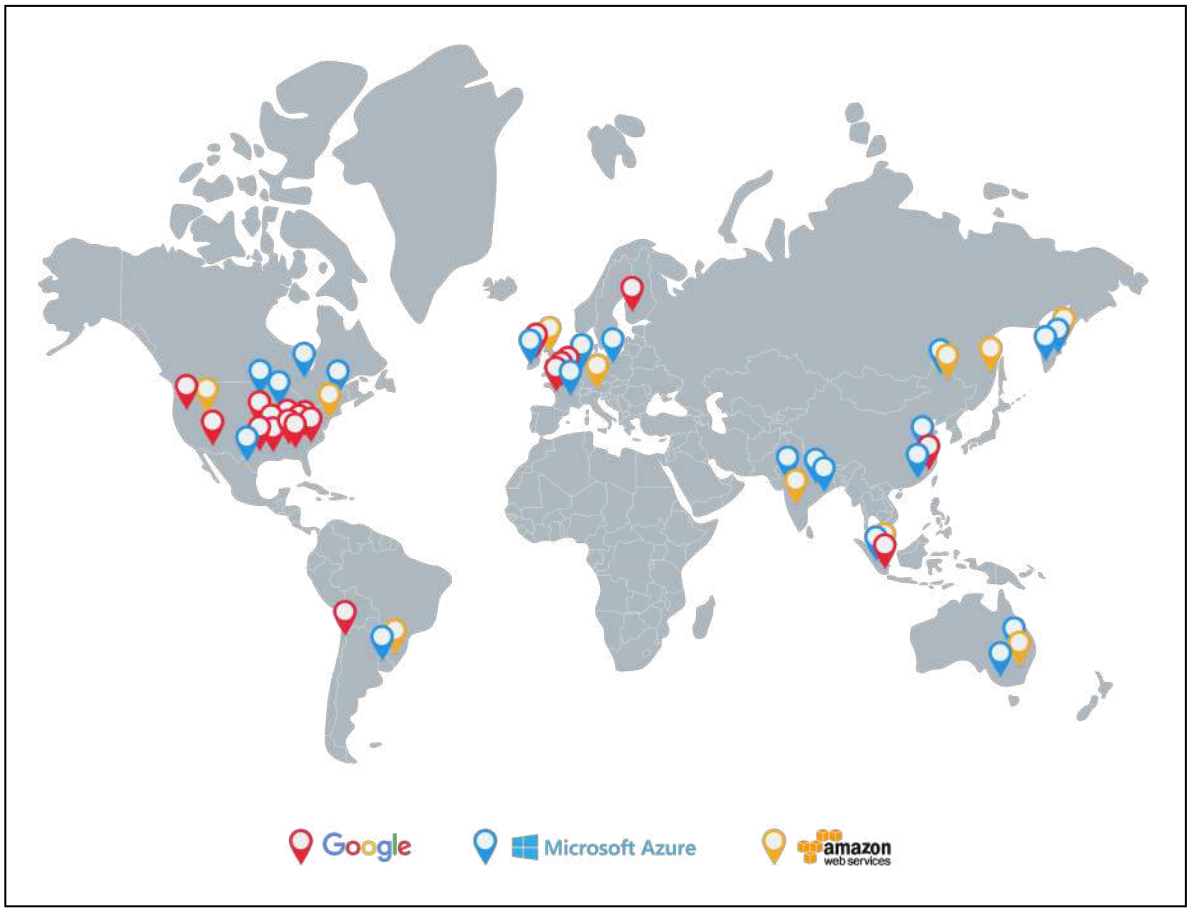


The example above is relatively simple. In reality, complex problems are encountered since we have many data centers across the globe, and each has millions of servers. Due to a decreasing human administrators to servers ratio, it's often not feasible to manually find the problems. Having a monitoring system reduces operational costs and encourages an automated way to detect failures.

There are fault-tolerant system designs that hide most of the failures from the end users, but it's crucial to catch the failures before they snowball into a bigger problem. The unplanned outage in services can be costly. For example, in October 2021, Meta's applications were down for nearly nine hours, resulting in a loss of around $13 million per hour. Such losses emphasize the potential impact of outages.

The IT infrastructure is spread widely around the globe. The illustration below the next paragraph gives an overview of distributed data centers of major cloud providers across the globe, circa 2021. The data centers are connected through private or public networks. Monitoring the servers in geo-separated data centers is essential.

According to Amazon, on December 7, 2021, "At 7:30 AM PST, an automated activity to scale capacity of one of the AWS services hosted in the main AW network triggered an unexpected behavior from a large number of clients inside the internal network. This resulted in a large surge of connection activity that overwhelmed the networking devices between the internal network and the main AWS network, resulting in communication delays between these networks. These delays increased latency and errors for services communicating between these networks, resulting in even more connection attempts and retries. This led to persistent congestion and performance issues on the devices connecting the two networks." According to one estimate, the outage cost of Amazon was $66,240 per minute.

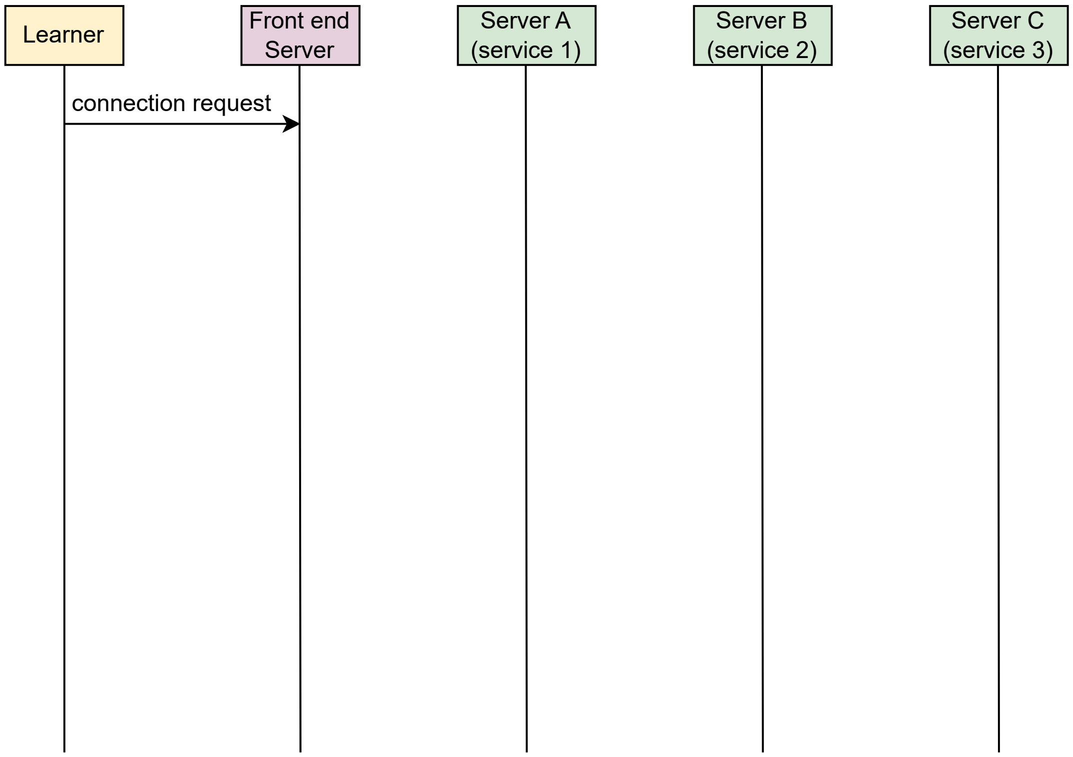


An overview of globally distributed data centers of AWS, Azure, and Google

Types of monitoring

Let's consider an example to understand the types of errors we want to monitor. At Educative, whenever a learner connects to an executable environment, a container is assigned. Consider service 1 in server A, whicl is responsible for allocating a container whenever a learner connects. Another service, 2 on server B takes this information and informs the servi responsible for UI. The UI service running in server C updates the UI for th learner. Let's assume that service 2 fails because of some error, and the learner sees the error of "Cannot connect..."

How do the Educative developers find out that a learner is facing this error?



The learner initiates a connection request to Educative

1 of 14

Now, what if a learner makes a request and it never reaches the servers of Educative. How will Educative know that a learner is facing an issue?

With the above examples, we can divide our monitoring focus into two broad categories of errors:

- **Service-side errors**: These are errors that are usually visible to monitoring services as they occur on servers. Such errors are reported

as error 5xx in HTTP response codes.
- **Client-side errors**: These are errors whose root cause is on the client-side. Such errors are reported as error 4xx in HTTP response codes. Some client-side errors are invisible to the service when client requests fail to reach the service.

We'll explore how to design a monitoring service to handle both scenarios in the upcoming chapter Monitoring Server-side Errors and Monitoring Client-side Errors. We want our monitoring systems to analyze our globally distributed services. It allows a better understanding of the system's components and agility to detect and respond to faults.

← Back
System Design: Distributed Monitoring

Next →
Prerequisites of a Monitoring System

✓ Mark as Completed

Detailed Design of a Monitoring System

Learn the details of designing a monitoring system and understand its pros and cons.

We'll cover the following

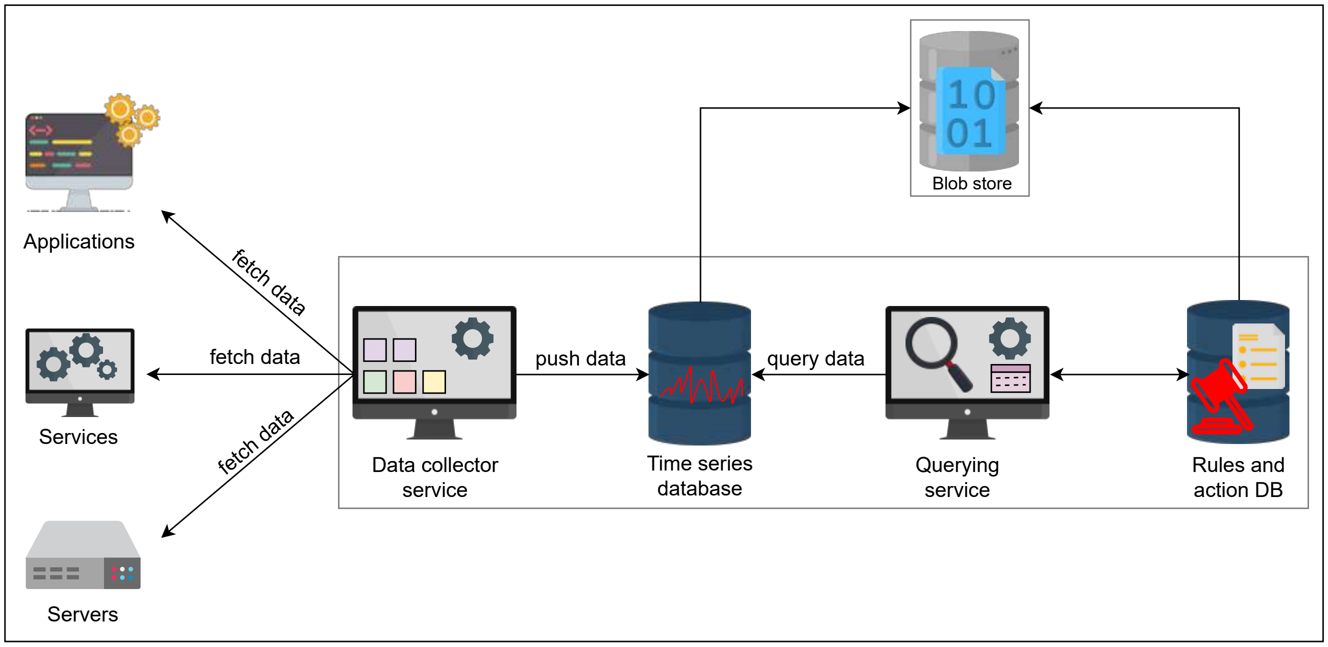
We'll discuss the core components of our monitoring system, identify the shortcomings of our design, and improve the design to fulfill our requirements.

Storage

We'll use time-series databases to save the data locally on the server where our monitoring service is running. Then, we'll integrate it with a separate storage node. We'll use blob storage to store our metrics.

We need to store metrics and know which action to perform if a metric has reached a particular value. For example, if CPU usage exceeds 90%, we generate an alert to the end user so the alert receiver can do take the necessary steps, such as allocate more resources to scale. For this purpose, we need another storage area that will contain the rules and actions. Let's call it a rules databse. Upon any violation of the rules, we can take appropriate action.

Here, we have identified two more components in our design—that is, a rules and action database and a storage node (a blob store).



Adding blob storage and a rules and action database

Data collector

We need a monitoring system to update us about our several data centers. We can stay updated if the information about our processes reaches us, which is possible through logging. We'll choose a pull strategy. Then, we'llextract our relevant metrics from the logs of the application. As discussed in our logging design, we used a distributed messaging queue. The message in the queue has the service name, ID, and a short description of t log. This will help us identify the metric and its information for a specific service. Exposing the relevant metrics to the data collector is necessary for monitoring any service so that our data collector can get the metrics from the service and store them into the time-series database.

A real-world example of a monitoring system based on the pull-based approach is DigitalOcean. It monitors millions of machines that are dispersed globally.

Point to Ponder

Question

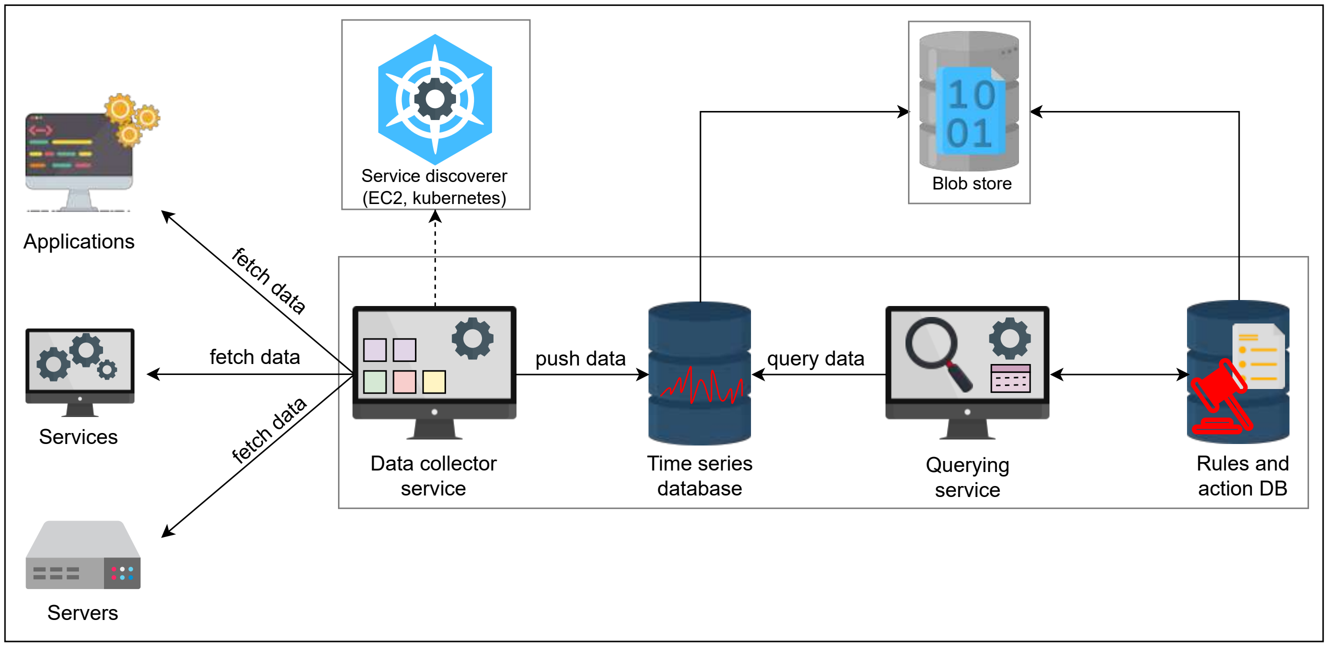
What are some drawbacks of using a push-based approach?

Show Answer

Service discoverer

The data collector is responsible for fetching metrics from the services it monitors. This way, the monitoring system doesn't need to keep a track of services. Instead, it can find them using discoverer service. We'll save the relative information of the services we have to monitor. We'll use a service discovery solution and integrate with several platforms and tools, including EC2, Kubernetes, and Consul. This will allow us to discover which services we have to monitor. Similar dynamic discovery can be used for newly commissioned hardware.

Let's add our newly identified component to our existing design.



Adding the service discoverer

Querying service

We want a service to access the database and fetch the relevant query results. We need this because we want to view the errors like values of a particular node's memory usage, or send an alert if a metric exceeds the set limit. Let's add the two components we need along with querying.

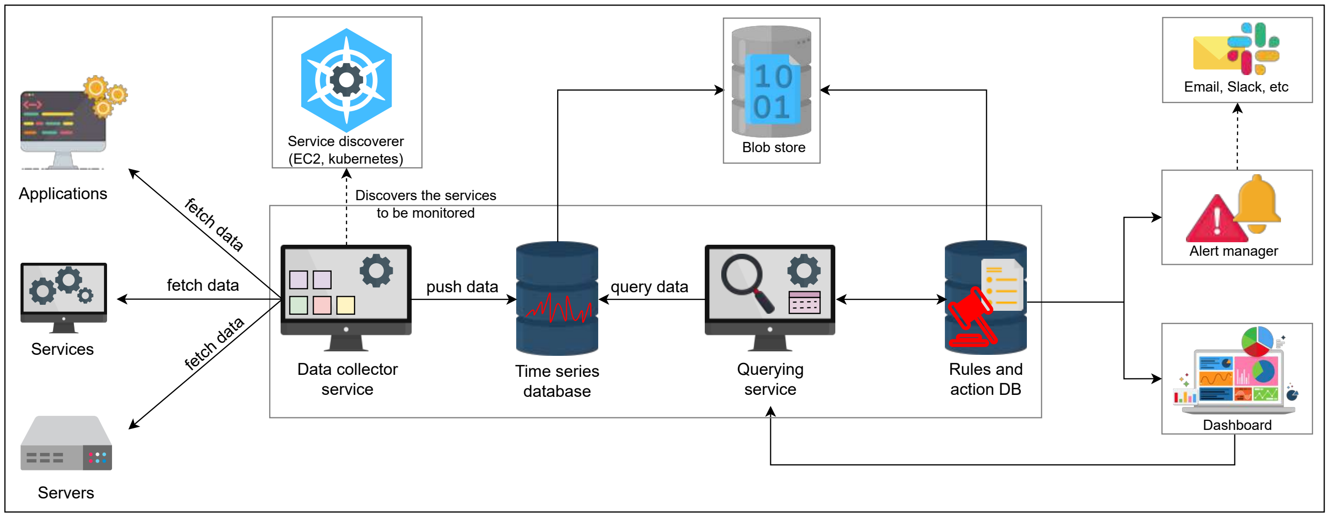
Alert Manager

The alert manager is responsible for sending alerts upon violations of set rules. It can send alerts as an email, a Slack message, and so on.

Dashboard

We can set dashboards by using the collected metrics to display the required information—for example, the number of requests in the current week.

Let's add the components discussed above, which completes our design of the monitoring system.



Detailed design of monitoring system

Our all-in-one monitoring service works for actively tracking systems and services. It collects and stores data, and it supports searches, graphs, and

Pros

- The design of our monitoring service ensures the smooth working of the operations and keeps an eye on signs of impending problems.
- Our design avoids overloading the network traffic by fetching the data itself.
- The monitoring service provides higher availability.

Cons

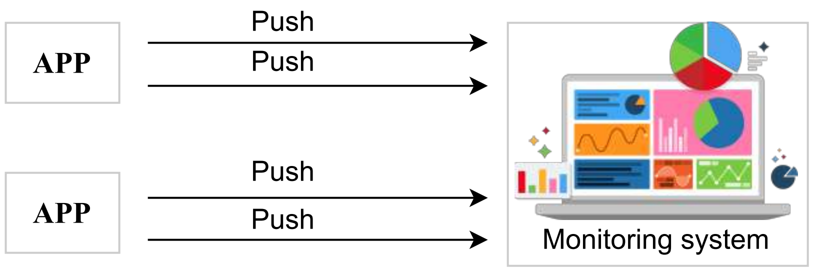
- The system seems scalable, but managing more servers to monitor can be a problem. For example, we have a dedicated server responsible for running the monitoring service. It can be a single point of failure (SPOF). To cater to SPOF, we can have a failover server for our monitoring system. Then, we also need to maintain consistency between actual and failover servers. However, such a design will also hit a scalability ceiling as the number of servers further increase.

- Monitoring collects an enormous amount of data 24/7, and keeping it forever might not be feasible. We need a policy and mechanisms to delete unwanted data periodically to efficiently utilize the resources.

Let’s think of a way to overcome the problems with our monitoring service.

Improving our design

We want to improve our design so that our system can scale better and decide what data to keep and what to delete. Let’s see how the push-based approach works. In a push-based approach, the application pushes its data to the monitoring system.



Push-based monitoring system

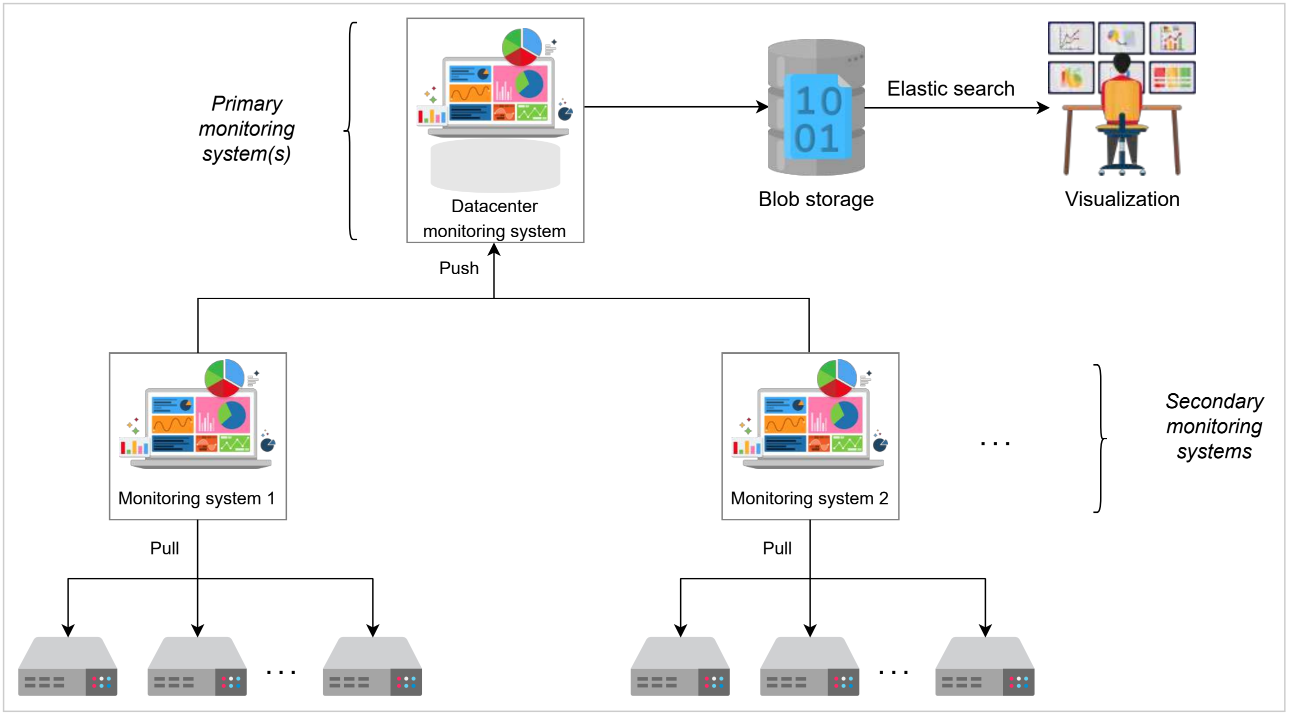
We used a pull-based strategy to avoid network congestion. This also allows the applications to be free of the aspect that they have to send the relevant monitoring data of to the system. Instead, the monitoring system fetches or pulls the data itself. To cater to scaling needs, we need to apply a push-based approach too. We’ll use a hybrid approach by combining our pull-based strategy with the push-based strategy.

We’ll keep using a pull-based strategy for several servers within a data center. We’ll also assign several monitoring servers for hundreds or thousands of servers within a data center—let’s say one server monitoring 5,000 servers. We’ll call them secondary monitoring servers.

Now, we’ll apply the push-based strategy. The secondary monitoring syster will push their data to a primary data center server. The primary data center server will push its data to a global monitoring service responsible for checking all the data centers spread globally.

We'll use blob storage to store our excessive data, apply elastic search, and view our relevant stats using a visualizer. As our servers or data centers increase, we'll add more monitoring systems. The design for this is given below.

> **Note**: Using a hierarchy of systems for scaling is a common design pattern in system design. By increasing nodes on a level or introducing additional levels in the hierarchy, we get the ability to scale according to our current needs.



Monitoring systems pull the data from various servers and then push the data to the data center monitoring system

1 of 2

Points to Ponder

Question 1

What happens if a local or global monitoring system is down?

Show Answer

1 of 2

Humans need to consume enormous amounts of data, and even after different kinds of data summarizations, the data can still be huge. Next, we'll tackle how to present enormous data to human administrators.

Back
Design of a Monitoring System

Next
Visualize Data in a Monitoring System

Background of Distributed Cache

Learn the fundamentals for designing a distributed cache.

We'll cover the following

The main goal of this chapter is to design a distributed cache. To achieve this goal, we should have substantial background knowledge, mainly on different reading and writing techniques. This lesson will help us build that background knowledge. Let’s look at the structure of this lesson in the table below

Structure of This Lesson

Section	Motivation

Writing policies	Data is written to cache and databases. The order in which data writi happens has performance implications. We'll discuss various writing policies to help decide which writing policy would be suitable for the distributed cache we want to design.
Eviction policies	Since the cache is built on limited storage (RAM), we ideally want to k the most frequently accessed data in the cache. Therefore, we'll discu different eviction policies to replace less frequently accessed data wi most frequently accessed data.
Cache invalidation	Certain cached data may get outdated. We'll discuss different invalid methods to remove stale or outdated entries from the cache in this se
	and what data structure to use for storage.
Cache client	A cache server stores cache entries, but a cache client calls the cache to request data. We'll discuss the details of a cache client library in th section.

Writing policies

Often, cache stores a copy (or part) of data, which is persistently stored in a data store. When we store data to the data store, some important questions arise:

> - Where do we store the data first? Database or cache?
> - What will be the implication of each strategy for consistency models?

The short answer is, it depends on the application requirements. Let's look at the details of different writing policies to understand the concept better:

- **Write-through cache**: The write-through mechanism writes on the cache as well as on the database. Writing on both storages can happen concurrently or one after the other. This increases the write latency but ensures strong consistency between the database and the cache.
- **Write-back cache**: In the write-back cache mechanism, the data is first written to the cache and asynchronously written to the database. Although the cache has updated data, inconsistency is inevitable in scenarios where a client reads stale data from the database. However, systems using this strategy will have small writing latency.
- **Write-around cache**: This strategy involves writing data to the database only. Later, when a read is triggered for the data, it's written to cache after a cache miss. The database will have updated data, but such a strategy isn't favorable for reading recently updated data.

Quiz

1

A system wants to write data and promptly read it back. At the same time, we want consistency between the cache and database. Which writing policy is the optimal choice?

A) Write-through cache

B) Write-around cache

C) Write-back cache

Submit Answer

Question 1 of 3
0 attempted

Reset Quiz

Eviction policies

One of the main reasons caches perform fast is that they're small. Small caches mean limited storage capacity. Therefore, we need an eviction mechanism to remove less frequently accessed data from the cache.

Several well-known strategies are used to evict data from the cache. The most well-known strategies include the following:

- Least recently used (LRU)
- Most recently used (MRU)
- Least frequently used (LFU)
- Most frequently used (MFU)

Other strategies like first in, first out (FIFO) also exist. The choice of each of these algorithms depends on the system the cache is being developed for.

Data Temperatures

Cache invalidation

Apart from the eviction of less frequently accessed data, some data residin in the cache may become stale or outdated over time. Such cache entries ar invalid and must be marked for deletion.

The situation demands a question: How do we identify stale entries?

Resolution of the problem requires storing metadata corresponding to each cache entry. Specifically, maintaining a time-to-live (TTL) value to deal with outdated cache items.

We can use two different approaches to deal with outdated items using TTL:

- **Active expiration**: This method actively checks the TTL of cache entries through a daemon process or thread.
- **Passive expiration**: This method checks the TTL of a cache entry at the time of access.

Each expired item is removed from the cache upon discovery.

Storage mechanism

Storing data in the cache isn't as trivial as it seems because the distributed cache has multiple cache servers. When we use multiple cache servers, the following design questions need to be answered:

- Which data should we store in which cache servers?
- What data structure should we use to store the data?

The above two questions are important design issues because they'll decide the performance of our distributed cache, which is the most important requirement for us. We'll use the following techniques to answer the questions above.

Hash function

It's possible to use hashing in two different scenarios:

- Identify the cache server in a distributed cache to store and retrieve data.
- Locate cache entries inside each cache server.

For the first scenario, we can use different hashing algorithms. However, consistent hashing or its flavors usually perform well in distributed systems because simple hashing won't be ideal in case of crashes or scaling.

In the second scenario, we can use typical hash functions to locate a cache entry to read or write inside a cache server. However, a hash function alone can only locate a cache entry. It doesn't say anything about managing data within the cache server. That is, it doesn't say anything about how to implement a strategy to evict less frequently accessed data from the cache server. It also doesn't say anything about what data structures are used to store the data within the cache servers. This is exactly the second design question of the storage mechanism. Let's take a look at the data structure next.

Linked list

We'll use a doubly linked list. The main reason is its widespread usage and simplicity. Furthermore, adding and removing data from the doubly linked list in our case will be a constant time operation. This is because we either evict a specific entry from the tail of the linked list or relocate an entry to the head of the doubly linked list. Therefore, no iterations are required.

> **Note:** Bloom filters are an interesting choice for quickly finding if a cache entry doesn't exist in the cache servers. We can use bloom filters to determine that a cache entry is definitely not present in the cache server, but the possibility of its presence is probabilistic. Bloom filters are quite useful in large caching or database systems.

Sharding in cache clusters

To avoid SPOF and high load on a single cache instance, we introduce sharding. Sharding involves splitting up cache data among multiple cache servers. It can be performed in the following two ways.

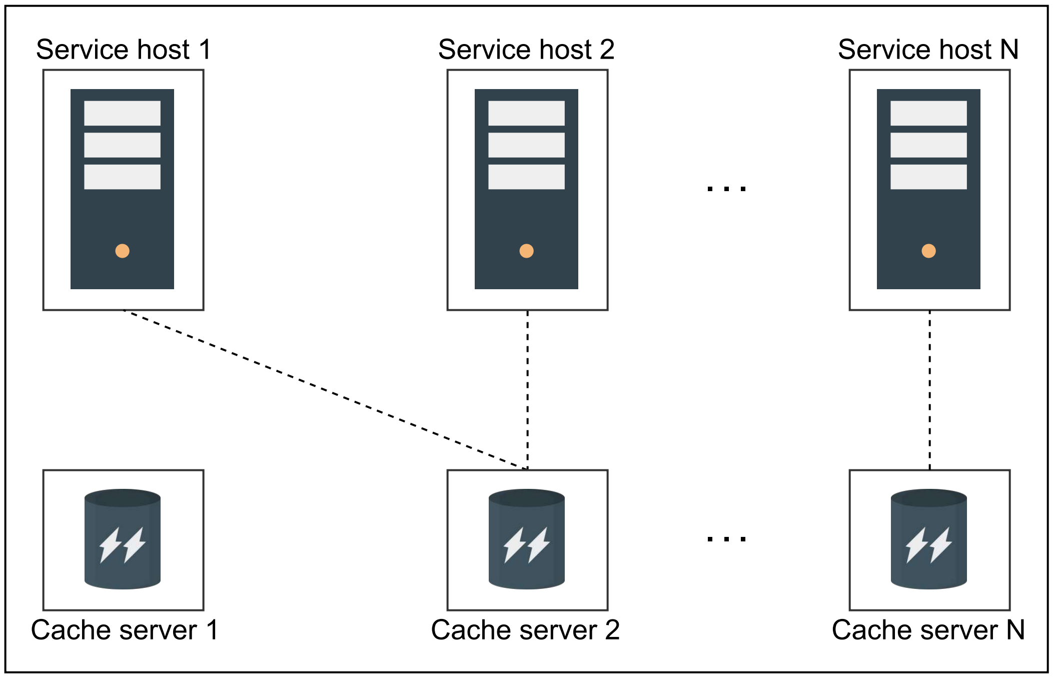
Dedicated cache servers

In the **dedicated cache servers** method, we separate the application and web servers from the cache servers.

The advantages of using dedicated cache servers are the following:

- There's flexibility in terms of hardware choices for each functionality.
- It's possible to scale web/application servers and cache servers separately.

Apart from the advantages above, working as a standalone caching service enables other microservices to benefit from them—for example, Cache as a Service. In that case, the caching system will have to be aware of different applications so that their data doesn't collide.

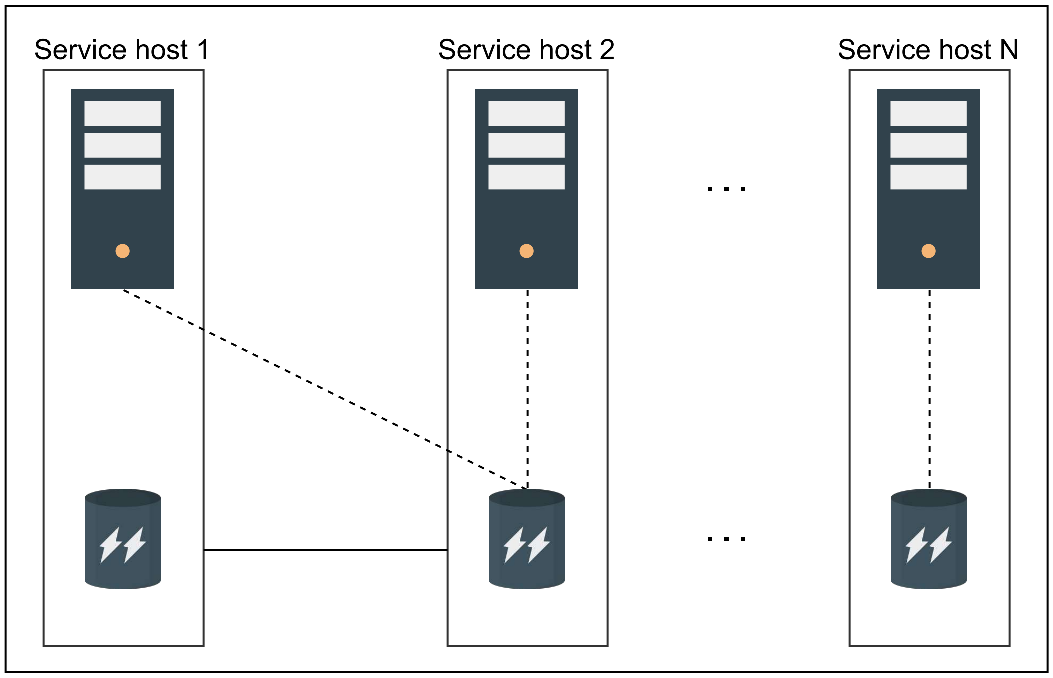


A depiction of service hosts coordinating with dedicated cache servers

Co-located cache

The co-located cache embeds cache and service functionality within the same host.

The main advantage of this strategy is the reduction in CAPEX and OPEX of extra hardware. Furthermore, with the scaling of one service, automatic scaling of the other service is obtained. However, the failure of one machine will result in the loss of both services simultaneously.



Hosting cache and application logic in the same machine

Cache client

We discussed that the hash functions should be used for the selection of cache servers. But what entity performs these hash calculations?

A cache client is a piece of code residing in hosting servers that do (hash) computations to store and retrieve data in the cache servers. Also, cache clients may coordinate with other system components like monitoring and configuration services. All cache clients are programmed in the same way that the same PUT, and GET operations from different clients return the same results. Some of the characteristics of cache clients are the following:

- Each cache client will know about all the cache servers.
- All clients can use well-known transport protocols like TCP or UDP to talk to the cache servers.

Point to Ponder

Question

What will be the behavior of cache clients to an access request if one of the cache servers is dead?

Show Answer

Conclusion

In this lesson, we learned what distributed caches are and highlighted their significance in distributed systems. We also discussed different storage and eviction mechanisms for caches. Caches are vital for any distributed system and are located at different points within the design of a system. It's important to understand how distributed caches can be designed as part of a large system.

← Back
System Design: The Distributed Cache

Next →
High-level Design of a Distributed C

✓ Mark as Complet

High-level Design of a Distributed Cache

Learn how we can develop a high-level design of a distributed cache.

We'll cover the following

In this lesson, we'll learn to design a distributed cache. We'll also discuss the trade-offs and design choices that can occur while we progress in our journey towards developing a solution.

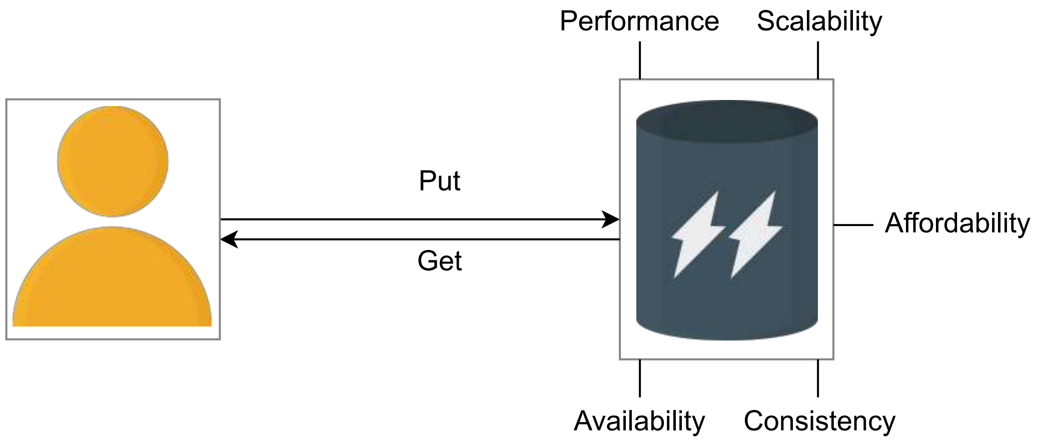
Requirements

Let us start by understanding the requirements of our solution.

Functional

The following are the functional requirements:

- **Insert data:** The user of a distributed cache system must be able to insert an entry to the cache.
- **Retrieve data:** The user should be able to retrieve data corresponding to a specific key.



Functional and non-functional requirements of a distributed cache

Non-functional requirements

We'll consider the following non-functional requirements:

- **High performance**: The primary reason for the cache is to enable fast retrieval of data. Therefore, both the `insert` and `retrieve` operations must be fast.
- **Scalability:** The cache system should scale horizontally with no bottlenecks on an increasing number of requests.
- **High availability**: The unavailability of the cache will put an extra burden on the database servers, which can also go down at peak load intervals. We also require our system to survive occasional failures of components and network, as well as power outages.

- **Consistency:** Data stored on the cache servers should be consistent. For example, different cache clients retrieving the same data from different cache servers (primary or secondary) should be up to date.
- **Affordability:** Ideally, the caching system should be designed from commodity hardware instead of an expensive supporting component within the design of a system.

API design

The API design for this problem is sufficiently easy since there are only two basic operations.

Insertion

The API call to perform insertion should look like this:

```
insert(key, value)
```

Parameter	Description
`key`	This is a unique identifier.
`value`	This is the data stored against a unique `key` .

This function returns an acknowledgment or an error depicting the problem at the server end.

Retrieval

The API call to retrieve data from the cache should look like this:

```
retrieve(key)
```

Parameter	Description
`key`	This returns the data stored against the `key`.

This call returns an object to the caller.

Point to Ponder

Question

The API design of the distributed cache looks exactly like the key-value store. What are the possible differences between a key-value store and a distributed cache?

Show Answer

Design considerations

Before designing the distributed cache system, it's important to consider some design choices. Each of these choices will be purely based on our application requirements. However, we can highlight some key differences here:

Storage hardware

If our data is large, we may require sharding and therefore use shard servers for cache partitions. Should these shard servers be specialized or commodity hardware? Specialized hardware will have good performance and storage capacity, but it will cost more. We can build a large cache from commodity servers. In general, the number of shard servers will depend on the cache's size and access frequency.

Furthermore, we can consider storing our data on the secondary storage of these servers for persistence while we still serve data from RAM. Secondary storage may be used in cases where a reboot happens, and cache rebuilding takes a long time. Persistence, however, may not be a requirement in a cache system if there's a dedicated persistence layer, such as a database.

Data structures

A vital part of the design has to be the speed of accessing data. Hash tables are data structures that take a constant time on average to store and retrieve data. Furthermore, we need another data structure to enforce an eviction algorithm on the cached data. In particular, linked lists are a good option (as discussed in the previous lesson).

Also, we need to understand what kind of data structures a cache can store. Even though we discussed in the API design section that we'll use strings for simplicity, it's possible to store different data structures or formats, like hash maps, arrays, sets, and so on, within the cache. In the next lesson, we'll see a practical example of such a cache.

Cache client

It's the client process or library that places the `insert` and `retrieve` calls. The location of the client process is a design issue. For example, it's possible to place the client process within a serving host if the cache is for internal use only. Otherwise, in the case where the caching system is provided as a service for external use, a dedicated cache client can send users' requests to the cache servers.

Writing policy

The writing strategy over the cache and database has consistency implications. In general, there’s no optimal choice, but depending on our application, the preference of writing policy is significantly important.

Eviction policy

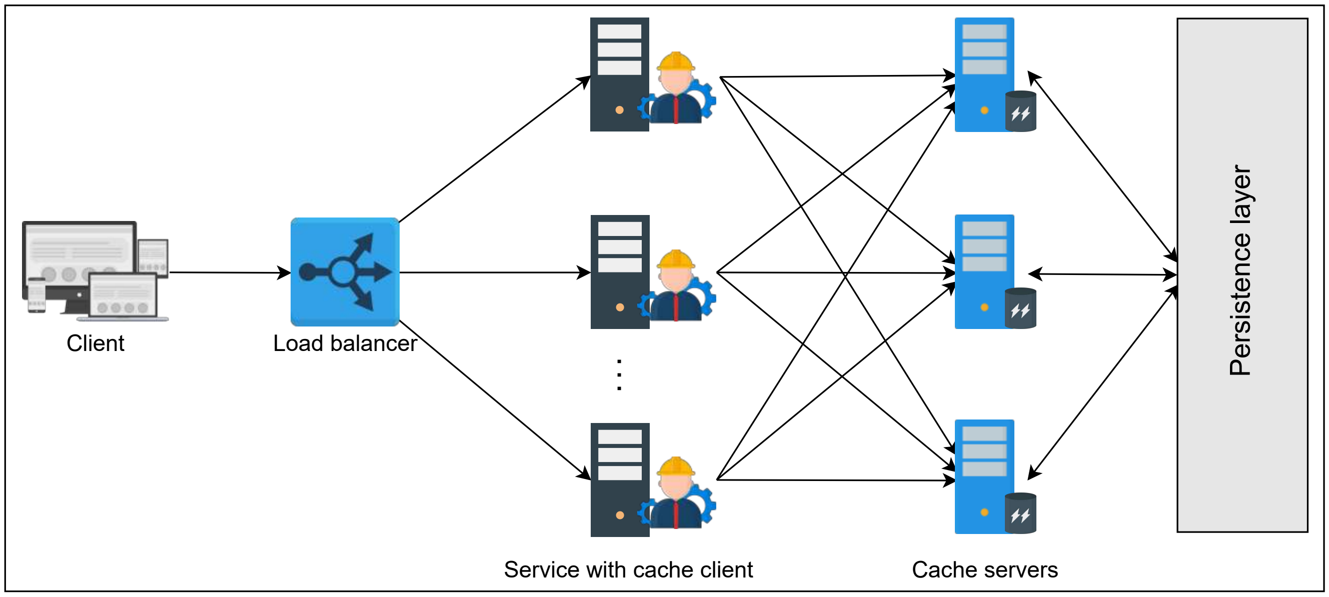
By design, the cache provides low-latency reads and writes. To achieve this, data is often served from RAM memory. Usually, we can’t put all the data in the cache due to the limited size of the cache as compared to the full dataset. So, we need to carefully decide what stays in the cache and how to make room for new entries.

With the addition of new data, some of the existing data may have to be evicted from the cache. However, choosing a victim entry depends on the eviction policy. Numerous eviction policies exist, but the choice again depends on the application using it. For instance, least recently used (LRU) can be a good choice for social media services where recently uploaded content will likely get the most views.

Apart from the details in the sections above, optimizing the time-to-live (TTL) value can play an essential role in reducing the number of cache misses.

High-level design

The following figure depicts our high-level design:



High-level design of distributed cache

The main components in this high-level design are the following:

- **Cache client**: This library resides in the service application servers. It holds all the information regarding cache servers. The cache client will choose one of the cache servers using a hash and search algorithm for each incoming `insert` and `retrieve` request. All the cache clients should have a consistent view of all the cache servers. Also, the resolution technique to move data to and from the cache servers should be the same. Otherwise, different clients will request different servers for the same data.
- **Cache servers**: These servers maintain the cache of the data. Each cache server is accessible by all the cache clients. Each server is connected to the database to store or retrieve data. Cache clients use TCP or UDP protocol to perform data transfer to or from the cache servers. However, if any cache server is down, requests to those serve are resolved as a missed cache by the cache clients.

← Back

Next -

Detailed Design of a Distributed Cache

Let's understand the detailed design of a distributed cache.

We'll cover the following

This lesson will identify some shortcomings of the high-level design of a distributed cache and improve the design to cover the gaps. Let's get started.

Find and remove limitations

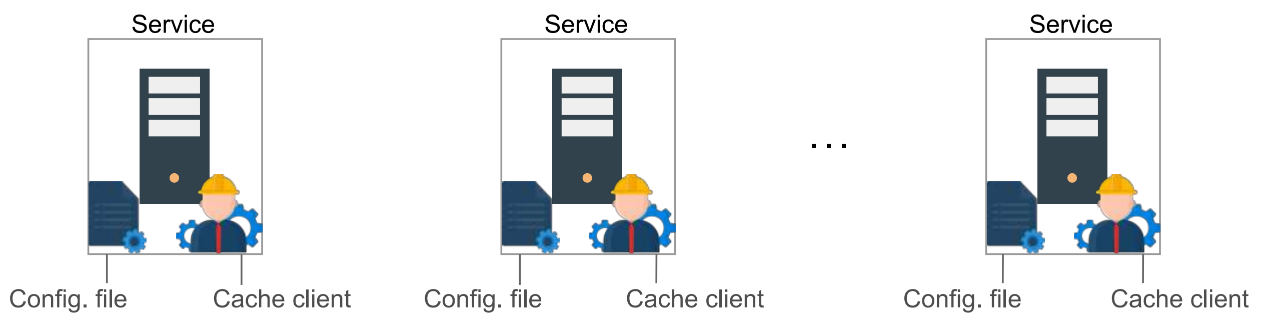
Before we get to the detailed design, we need to understand and overcome some challenges:

- There's no way for the cache client to realize the addition or failure of a cache server.
- The solution will suffer from the problem of single point of failure (SPOF) because we have a single cache server for each set of cache dat
  Not only that, if some of the data on any of the cache servers is frequently accessed (generally referred to as a hotkey problem), then our performance will also be slow.
- Our solution also didn't highlight the internals of cache servers. That i
  what kind of data structures will it use to store and what eviction policy

will it use?

Maintain cache servers list

Let's start by resolving the first problem. We'll take incremental steps toward the best possible solution. Let's look at the following slides to get an idea of each of the solutions described below:



[Solution 1] Maintaining a configuration file in each server that the cache client can use

1 of 3

- **Solution 1**: It's possible to have a configuration file in each of the service hosts where the cache clients reside. The configuration file will contain the updated health and metadata required for the cache clien to utilize the cache servers efficiently. Each copy of the configuration service can be updated through a push service by any DevOps tool. Th main problem with this strategy is that the configuration file will have to be manually updated and deployed through some DevOps tools.

- **Solution 2**: We can store the configuration file in a centralized location that the cache clients can use to get updated information about cache servers. This solves the deployment issue, but we still need to manually update the configuration file and monitor the health of each server.
- **Solution 3**: An automatic way of handling the issue is to use a configuration service that continuously monitors the health of the cache servers. In addition to that, the cache clients will get notified when a new cache server is added to the cluster. When we use this strategy, no human intervention or monitoring will be required in case of failures or the addition of new nodes. Finally, the cache clients obtain the list of cache servers from the configuration service.

> The configuration service has the highest operational cost. At the same time, it's a complex solution. However, it's the most robust among all the solutions we presented.

Improve availability

The second problem relates to cache unavailability if the cache servers fail. A simple solution is the addition of replica nodes. We can start by adding one primary and two backup nodes in a cache shard. With replicas, there's always a possibility of inconsistency. If our replicas are in close proximity, writing over replicas is performed synchronously to avoid inconsistencies between shard replicas. It's crucial to divide cache data among shards so that neither the problem of unavailability arises nor any hardware is wasted.

This solution has two main advantages:

- There's improved availability in case of failures.

- Hot shards can have multiple nodes (primary-secondary) for reads.

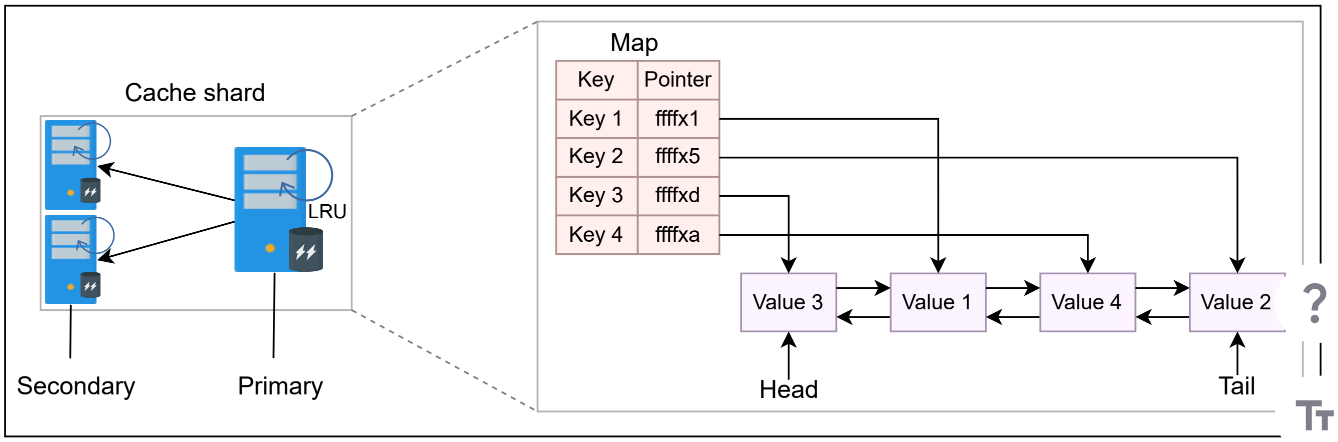
Not only will such a solution improve availability, but it will also add to the performance.

Internals of cache server

Each cache client should use three mechanisms to store and evict entries from the cache servers:

- **Hash map**: The cache server uses a hash map to store or locate different entries inside the RAM of cache servers. The illustration below shows that the map contains pointers to each cache value.
- **Doubly linked list**: If we have to evict data from the cache, we require a linked list so that we can order entries according to their frequency of access. The illustration below depicts how entries are connected using a doubly linked list.
- **Eviction policy**: The eviction policy depends on the application requirements. Here, we assume the least recently used (LRU) eviction policy.

A depiction of a sharded cluster along with a node's data structure is provided below:

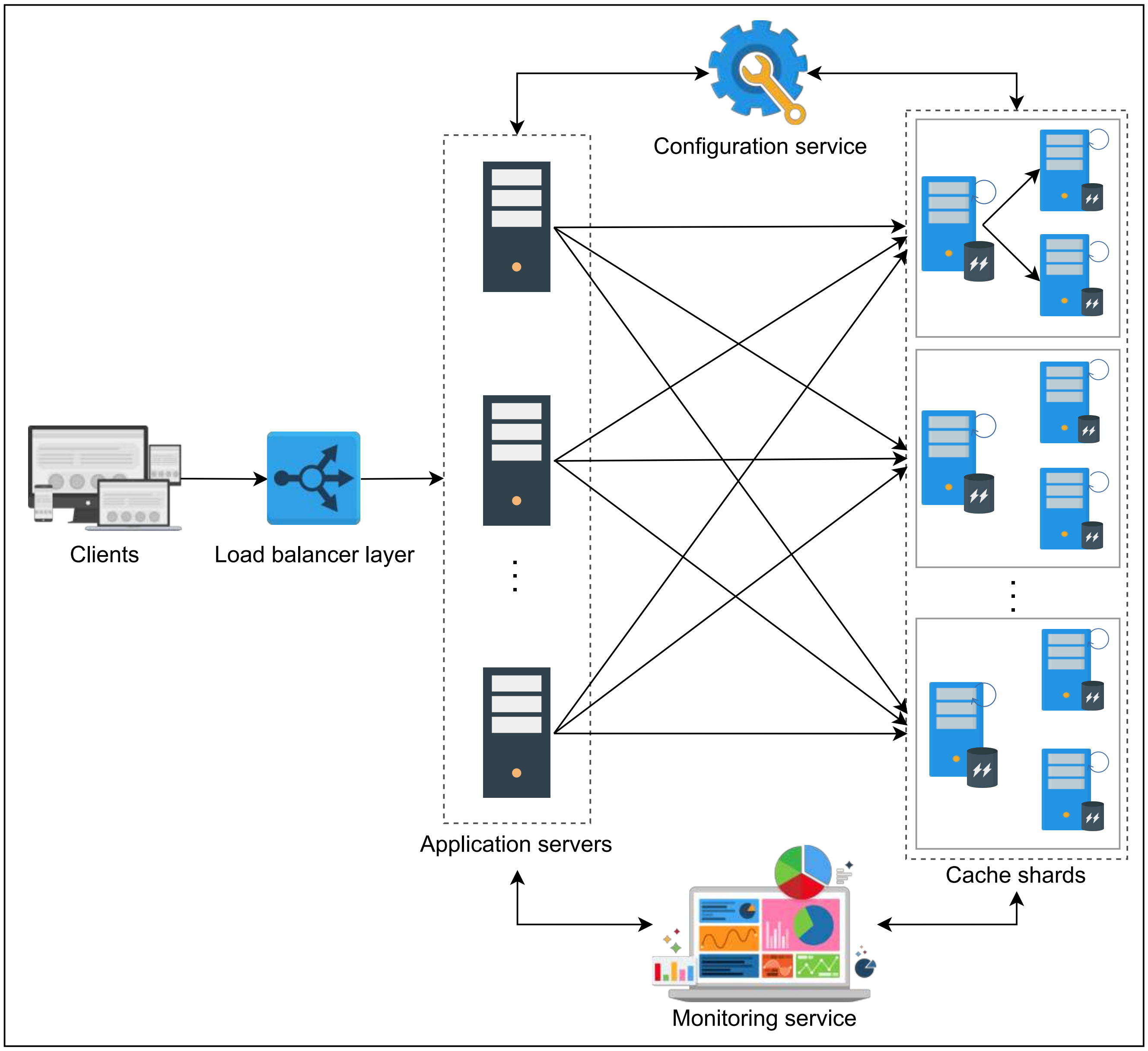


A shard primary and replica, each with the same internal mechanisms

> It’s evident from the explanation above that we don’t provide a `delete` API. This is because the eviction (through eviction algorithm) and deletion (of expired entries through TTL) is done locally at cache servers. Nevertheless, situations can arise where the `delete` API may be required. For example, when we delete a recently added entry from the database, it should result in the removal of items from the cache for the sake of consistency.

Detailed design

We’re now ready to formalize the detailed design after resolving each of the three previously highlighted problems. Look at the detailed design below:



Detailed design of a distributed caching system

Let’s summarize the proposed detailed design in a few points:

- The client’s requests reach the service hosts through the load balancers where the cache clients reside.
- Each cache client uses consistent hashing to identify the cache server. Next, the cache client forwards the request to the cache server maintaining a specific shard.
- Each cache server has primary and replica servers. Internally, every server uses the same mechanisms to store and evict cache entries.

- Configuration service ensures that all the clients see an updated and consistent view of the cache servers.
- Monitoring services can be additionally used to log and report different metrics of the caching service.

> **Note:** An important aspect of the design is that cache entries are stored and retrieved from RAM. We discussed the suitability of RAM for designing a caching system in the previous lesson.

Point to Ponder

Memcached versus Redis

Let's compare Memcached and Redis.

We'll cover the following

Introduction

This lesson will discuss some of the widely adopted real-world implementations of a distributed cache. Our focus will be on two well-known open-source frameworks: Memcached and Redis. They're highly scalable, highly performant, and robust caching tools. Both of these techniques follow the client-server model and achieve a latency of sub-millisecond. Let's discuss each one of them and then compare their usefulness.

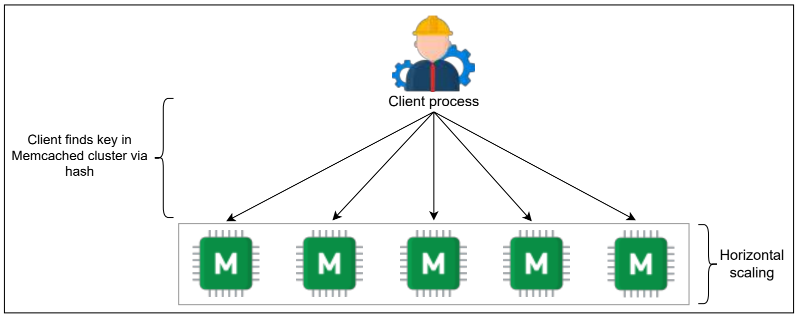
Memcached

Memcached was introduced in 2003. It's a key-value store distributed cache designed to store objects ve fast. Memcached stores data in the form of a key-value pair. Both the key and the value are strings.

This means that any data that has been stored will have to be serialized. So, Memcached doesn't support and can't manipulate different data structures.

Memcached has a client and server component, each of which is necessary to run the system. The system is designed in a way that half the logic is encompassed in the server, whereas the other half is in the client. However, each server follows the **shared-nothing architecture**. In this architecture, servers are unaware of each other, and there's no synchronization, data sharing, and communication between the servers.

Due to the disconnected design, Memcached is able to achieve almost a deterministic query speed $(O(1))$ serving millions of keys per second using a high-end system. Therefore, Memcached offers a high throughput and low latency.



Design of a typical Memcached cluster

As evident from the design of a typical Memcached cluster, Memcached scales well horizontally. The client process is usually maintained with the service host that also interacts with the authoritative storage (back-end database).

Facebook and Memcached

The data access pattern in Facebook requires frequent reads and updates because views are presented to the users on the fly instead of being generated ahead of time. Because Memcached is simple, it was an easy choice for the solution because Memcached started developing in 2003 whereas Facebook was developed in 2004. In fact, in some cases, Facebook and Memcached teams worked together to find solutions.

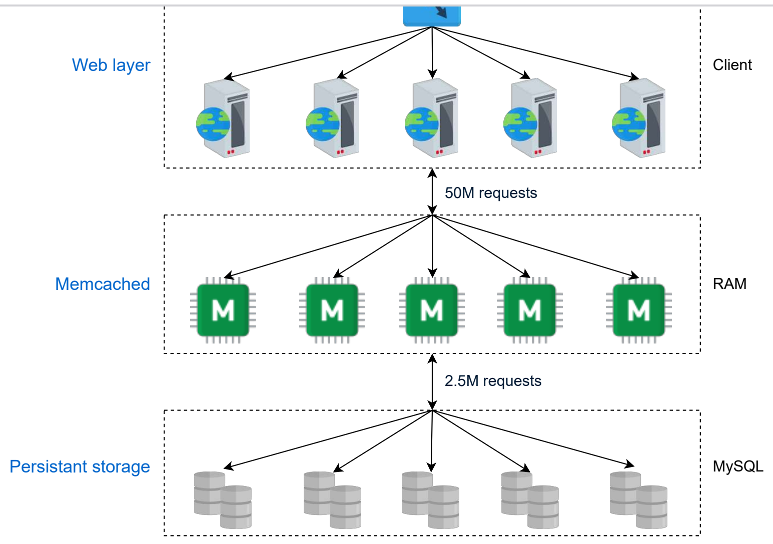
What about Redis?

Some of the simple commands of Memcached include the following:

```
get <key_1> <key_2> <key_3> ...
set <key> <value> ...
delete <key>[<time>] ...
```

At Facebook, Memcached sits between the MySQL database and the web layer that uses roughly 28 TeraBytes of RAM spread across more than 800 servers (as of 2013). By an approximation of least recently used (LRU) eviction policy, Facebook is able to achieve a cache hit rate of 95%.

The following illustration shows the high-level design of caching architecture at Facebook. As we can see, out of a total of 50 million requests made by the web layer, only 2.5 million requests reach the persistence layer.



Facebook using a layer of Memcached sitting between persistence and web layer

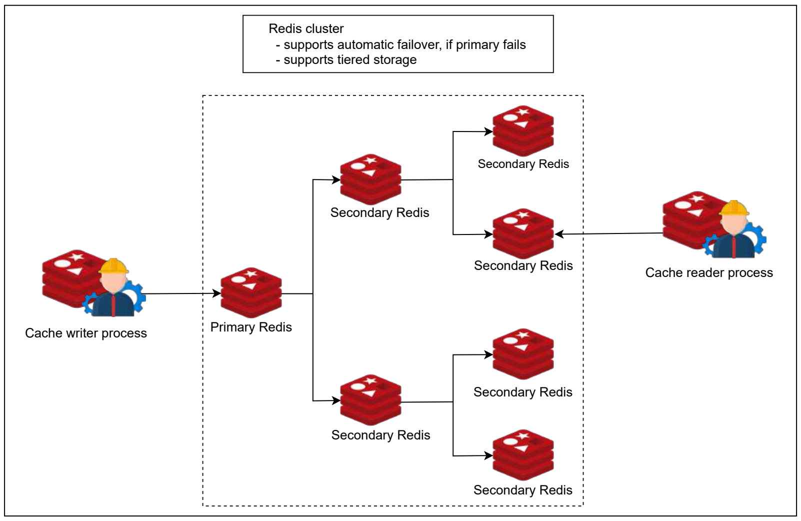
Redis

Redis is a data structure store that can be used as a cache, database, and message broker. It offers rich features at the cost of additional complexity. It has the following features:

- **Data structure store**: Redis understands the different data structures it stores. We don't have to retrieve data structures from it, manipulate them, and then store them back. We can make in-house changes that save both time and effort.
- **Database**: It can persist all the in-memory blobs on the secondary storage.
- **Message broker**: Asynchronous communication is a vital requirement in distributed systems. Redis can translate millions of messages per second from one component to another in a system.

Redis provides a built-in replication mechanism, automatic failover, and different levels of persistence. Apart from that, Redis understands

Memcached protocols, and therefore, solutions using Memcached can translate to Redis. A particularly good aspect of Redis is that it separates data access from cluster management. It decouples data and controls the plane. This results in increased reliability and performance. Finally, Redis doesn't provide strong consistency due to the use of asynchronous replication.

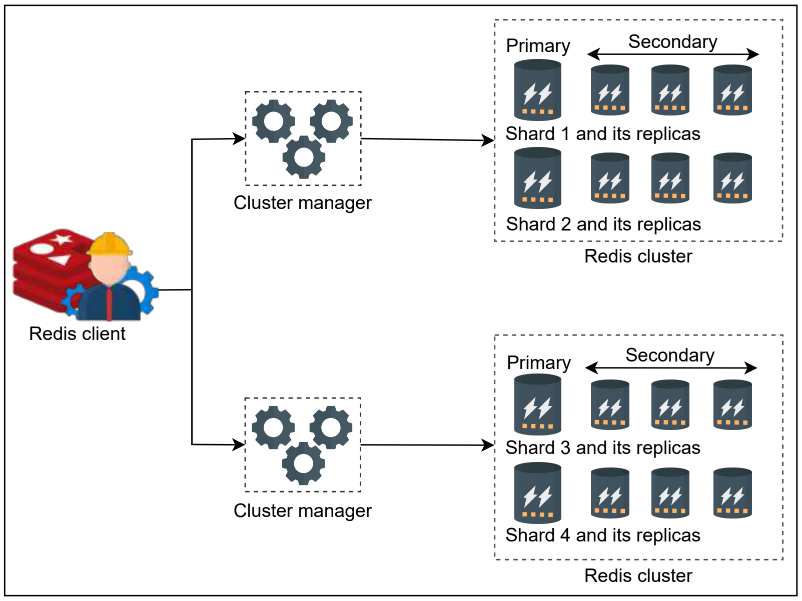


Redis structure supporting automatic failover using redundant secondary replicas

Redis cluster

Redis has built-in cluster support that provides high availability. This is called Redis Sentinel. A cluster has one or more Redis databases that are queried using multithreaded proxies. Redis clusters perform automatic sharding where each shard has primary and secondary nodes. However, the number of shards in a database or node is configurable to meet the expectations and requirements of an application.

Each Redis cluster is maintained by a cluster manager whose job is to detect failures and perform automatic failovers. The management layer consists of monitoring and configuration software components.

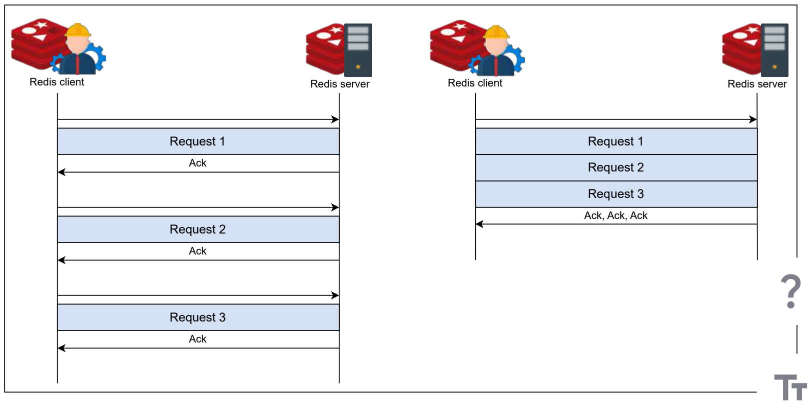


Architecture of Redis clusters

Pipelining in Redis

Since Redis uses a client-server model, each request blocks the client until the server receives the result. A Redis client looking to send subsequent requests will have to wait for the server to respond to the first request. So, the overall latency will be higher.

Redis uses **pipelining** to speed up the process. **Pipelining** is the process of combining multiple requests from the client side without waiting for a response from the server. As a result, it reduces the number of RTT spans for multiple requests.



Redis client-server communication without pipelining versus Redis client-server communication with pipelining

The process of pipelining reduces the latency through RTT and the time to do socket level I/O. Also, mode switching through system calls in the operating system is an expensive operation that's reduced significantly via pipelining. Pipelining the commands from the client side has no impact on how the server processes these requests.

For example, two requests pipelined by the client reach the server, and the server can't entertain the second. The server provides a result for the first and returns an error for the second. The client is independent in batching similar commands together to achieve maximum throughput.

> **Note:** Pipelining improves the latency to a minimum of five folds if both the client and server are on the same machine. The request is sent on a loopback address (`127.0.0.1`). The true power of pipelining is highlighted in systems where requests are sent to distant machines.

Memcached versus Redis

Even though Memcached and Redis both belong to the NoSQL family, there are subtle aspects that set them apart:

- **Simplicity:** Memcached is simple, but it leaves most of the effort for managing clusters left to the developers of the cluster. This, however, means finer control using Memcached. Redis, o the other hand, automates most of the scalability and data division tasks.

- **Persistence:** Redis provides persistence by properties like append only file (AOF) and Redis database (RDB) snapshot. There's no persistence support in Memcached. But this limitation can be catered to by using third-party tools.
- **Data types**: Memcached stores objects, whereas Redis supports strings, sorted sets, hash maps, bitmaps, and hyper logs. However, the maximum key or value size is configurable.
- **Memory usage:** Both tools allow us to set a maximum memory size for caching. Memcached uses the slab allocation method for reducing fragmentation. However, when we update the existing entries' size or store many small objects, there may be a wastage of memory. Nonetheless, there are configuration workarounds to resolve such issues.
- **Multithreading:** Redis runs as a single process using one core, whereas Memcached can efficiently use multicore systems with multithreading technology. We could argue that Redis was designed to be a single-threaded process that reduces the complexity of multithreaded systems. Nonetheless, multiple Redis processes can be executed for concurrency. At the same time, Redis has improved over the years by tweaking its performance. Therefore, Redis can store small data items efficiently. Memcached can be the right choice for file sizes above 100 K.
- **Replication:** As stated before, Redis automates the replication process via few commands, whereas replication in Memcached is again subject to the usage of third-party tools. Architecturally, Memcached can scale well horizontally due to its simplicity. Redis provides

scalability through clustering that's considerably complex.

The table below summarizes some of the main differences and common features between Memcached and Redis:

Features Offered by Memcached and Redis

Feature	Memcached
Low latency	Yes
Persistence	Possible via third-party too
Multilanguage support	Yes
Data sharding	Possible via third-party too
Ease of use	Yes
Multithreading support	Yes
Support for data structure	Objects
Support for transaction	No
Eviction policy	LRU
Lua scripting support	No
Geospatial support	No

To summarize, Memcached is preferred for smaller, simpler read-heavy systems, whereas Redis is useful for systems that are complex and are both read- and write-heavy.

Points to Ponder

Question 1

Based on the implementation details, which of the two frameworks (Memcached or Redis) has a striking similarity with the distributed cache that we designed in the previous lesson?

Show Answer

1 of 3

Conclusion

It's impossible to imagine high-speed large-scale solutions without the use of a caching system. In this chapter, we covered the need for a caching system and its fundamental details, and we orchestrated a basic distributed cache system. We also familiarized ourselves with the design and features of two of the most well-known caching frameworks

Requirements of a Distributed Messaging Queue’s Design

Learn about the requirements of designing a distributed messaging queue using a strawman solution.

We'll cover the following

Requirements

In a **distributed messaging queue**, data resides on several machines. Our aim is to design a distributed messaging queue that has the following functional and non-functional requirements.

Listed below are the actions that a client should be able to perform:

- **Queue creation:** The client should be able to create a queue and set some parameters—for example, queue name, queue size, and **maximum message size.**↴
- **Send message:** Producer entities should be able to send messages to a queue that’s intended for them.

- **Receive message:** Consumer entities should be able to receive messages from their respective queues.
- **Delete message:** The consumer processes should be able to delete a message from the queue after a successful processing of the message.
- **Queue deletion:** Clients should be able to delete a specific queue.

Non-functional requirements

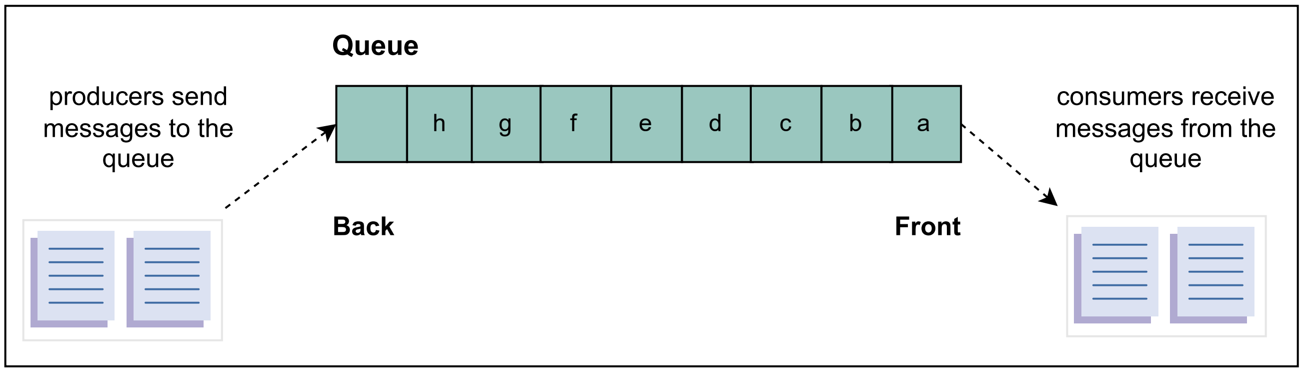
Our design of a distributed messaging queue should adhere to the following non-functional requirements:

- **Durability**: The data received by the system should be durable and shouldn't be lost. Producers and consumers can fail independently, and a queue with data durability is critical to make the whole system work, because other entities are relying on the queue.
- **Scalability**: The system needs to be scalable and capable of handling the increased load, queues, producers, consumers, and the number of messages. Similarly, when the load reduces, the system should be able to shrink the resources accordingly.
- **Availability**: The system should be highly available for receiving and sending messages. It should continue operating uninterrupted, even after the failure of one or more of its components.
- **Performance:** The system should provide high throughput and low latency.

Single-server messaging queue

Before we embark on our journey to map out the design of a distributed messaging queue, we should recall how queues are used within a single server where the producer and consumer processes are also on the same node. A producer or consumer can access a single-server queue by acquiri the locking mechanism to avoid data inconsistency. The queue is considered a critical section where only one entity, either the producer or consumer, can access the data at a time.

However, several aspects restrain us from using the single-server messaging queue in today's distributed systems paradigm. For example, it becomes unavailable to cooperating processes, producers and consumers, in the event of hardware or network failures. Moreover, performance takes a major hit as contention on the lock increases. Furthermore, it is neither scalable nor durable.


Queue
producers send messages to the queue
h
g
f
e
d
c
b
a
consumers receive messages from the queue
Back
Front

Multiple producers and consumers interact via a single messaging queue

Point to Ponder

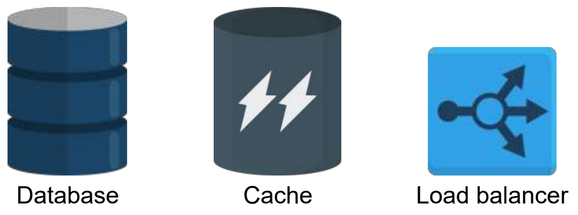
Question

Can we extend the design of a single-server messaging queue to a distributed messaging queue?

Show Answer

Building blocks we will use

The design of a distributed messaging queue utilizes the following building blocks:



The building blocks used to design a distributed messaging queue

- **Database(s)** will be required to store the metadata of queues and users.
- **Caches** are important to keep frequently accessed data, whether it be data pertaining to users or queues metadata.
- **Load balancers** are used to direct incoming requests to servers where the metadata is stored.

In our discussion on messaging queues, we focused on their functional and non-functional requirements. Before moving on to the process of designing a distributed messaging queue, it's essential for us to discuss some key considerations and challenges that may affect the design.

← Back
System Design: The Distributed Messaging ...

Next →
Considerations of a Distributed Messaging Q...

✓ Mark as Completed

Considerations of a Distributed Messaging Queue’s Design

Learn about the factors that affect the design of a messaging queue.

We'll cover the following

Before embarking on our journey to design a distributed messaging queue, let’s discuss some major factors that could significantly affect the design. These include the order of messages, the effect of the ordering on performance, and the management of concurrent access to the queue. We discuss each of these factors in detail below.

Ordering of messages

A messaging queue is used to receive messages from producers. These messages are consumed by the consumers at their own pace. Some operations are critical in that they require strict ordering of the execution of the tasks, driven by the messages in the queue. For example, while chatting over a messenger application with a friend, the messages should be delivered in order; otherwise, such communication can be confusing, to say the least. Similarly, emails received by a user from different users may not require strict ordering. Therefore, in some cases, the strict order of incoming messages in the queue is essential, while many use cases can tolerate some reordering.

Let's discuss the following two categories of messages ordering in a queue:

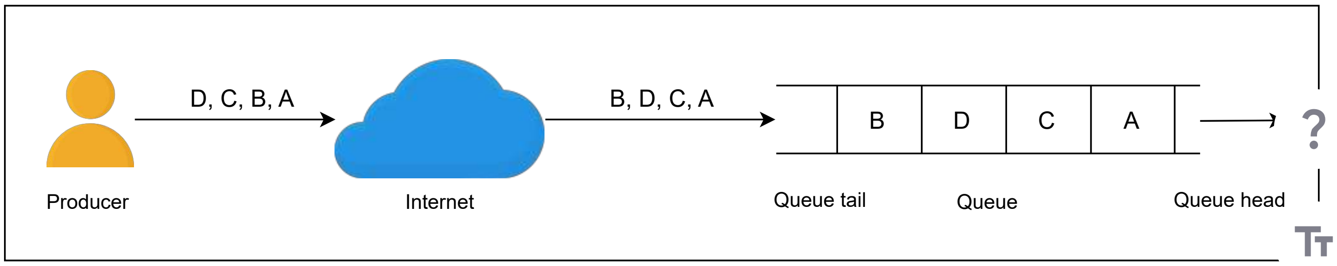
- *Best-effort ordering*
- *Strict ordering*

How is the order associated with messages?

Best-effort ordering

With the **best-effort ordering** approach, the system puts the messages in a specified queue in the same order that they're received.

For example, as shown in the following figure, the producer sends four messages, A, B, C, and D, in the same order as illustrated. Due to network congestion or some other issue, message B is received after message D. Hence, the order of messages is A, C, D, and B at the receiving end. Therefore, in this approach, the messages will be put in the queue in the same order they were received instead of the order in which they were produced on the client side.



Best-effort ordering: Messages are placed in a queue in the same order that they're received and not in the order they were sent

Strict ordering

The strict ordering technique preserves the ordering of messages more rigorously. Through this approach, messages are placed in a queue in the order that they're produced.

Before putting messages in a queue in the correct sequence, it's crucial to have a mechanism to identify the order in which the messages were produced on the client side. Often, a unique identifier or time-stamp is used to mark a message when it's produced.

Point to Ponder

Question

Who'll be responsible for providing the sequence numbers?

Show Answer

One of the following three approaches can be used for ordering incoming messages:

1. **Monotonically increasing numbers:** One way to order incoming messages is to assign monotonically increasing numbers to messages on the server side. When the first message arrives, the system assigns it a number, such as 1. It then assigns the number 2 to the second message, and so on.

However, there are potential drawbacks to this approach. First, when a burst of requests is received, it acts as a bottleneck that affects the system's performance because the system has to assign an ID in a specified sequence to a message while the other messages wait for their turn.

Second, it still doesn't tackle the problem that arises when a message is received before the one that's produced earlier at the client side. Because of this, it doesn't guarantee that it will generate the correct order for the messages produced at the client side.

2. **Causality-based sorting at the server side:** Keeping in view the drawbacks of using monotonically increasing numbers, another approach that can be used for time-stamping and ordering of incoming messages is causality-based sorting. In this approach, messages are sorted based on the time stamp that was produced at the client side and are put in a queue accordingly. The major drawback of this approach is that for multiple client sessions, the service can't determine the order in terms of wall-clock time.

3. **Using time stamps based on synchronized clocks:** To tackle the potential issues that arise with both of the approaches described above, we can use another appropriate method to assign time stamps to messages that's based on synchronized clocks. In this approach, the time stamp (ID) provided to each message through a synchronized clock is unique and in the correct sequence of production of messages. We can tag a unique process identifier with the time stamp to make the overall message identifier unique and tackle the situation when two concurrent sessions ask for a time stamp at the exact same time. Moreover, with this approach, the server can easily identify delayed messages based on the time stamp and wait for the delayed messages.

   As we discussed in the section on the sequencer building block, we car get sequence numbers that fulfill double duty as sequence numbers and globally synchronized wall clock time stamps. Using this approach, our service can globally order messages across client sessions as well.

To conclude, the most appropriate mechanism to provide a unique ID or time stamp to incoming messages, from among the three approaches described above, involves the use of synchronized clocks.

Sorting

Once messages are received at the server side, we need to sort them based on their time stamps. Therefore, we use an appropriate online sorting algorithm for this purpose.

Point to Ponder

Question

Suppose that a message sent earlier arrives late due to a network delay. What would be the proper approach to handle such a situation?

Show Answer

Effect on performance

Primarily, a queue is designed for first-in, **first-out (FIFO) operations**;. Fir
in, first-out operations suggest that the first message that enters a queue is always handed out first. However, it isn't easy to maintain this strict order distributed systems. Since message A was produced before message B, it's

still uncertain that message A will be consumed before message B. Using monotonically increasing message identifiers or causality-bearing identifiers provide high throughput while putting messages in a queue. Though the need for the online sorting to provide a strict order takes some time before messages are ready for extraction. To minimize latency caused by the online sorting, we use a **time-window** approach.

requests to give out messages one by one. If that's not required, we have better throughput and lower latency at the receiving end.

Due to the reasons mentioned above, many distributed messaging queue solutions either don't guarantee a strict order or have limitations around throughput. As we saw previously, the queues have to perform many additional validations and coordination operations to maintain the order.

Managing concurrency

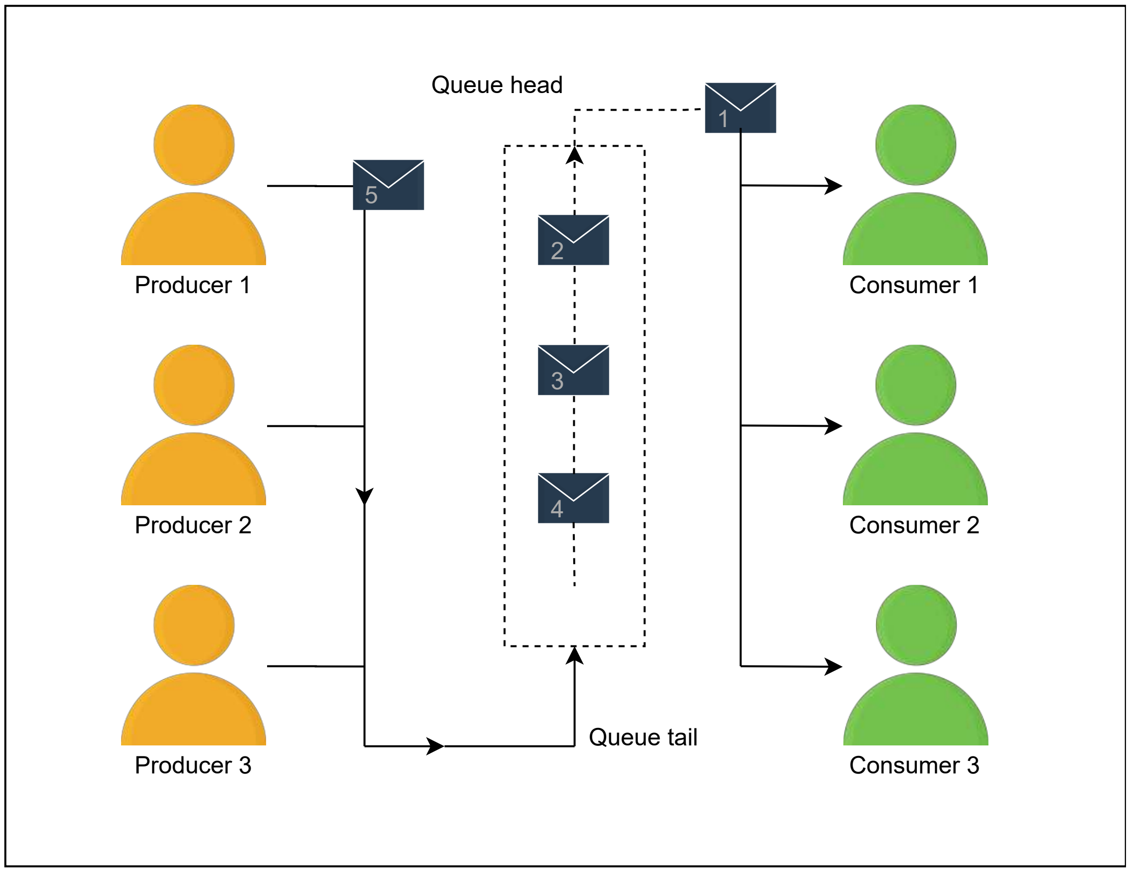
Concurrent queue access needs proper management. Concurrency can take place at the following stages:

- When multiple messages arrive at the same time.
- When multiple consumers request concurrently for a message.

The first solution is to use the locking mechanism. When a process or thread requests a message, it should acquire a lock for placing or consuming messages from the queue. However, as was discussed earlier, this approach has several drawbacks. It's neither scalable nor performant.

Another solution is to serialize the requests using the system's buffer at both ends of the queue so that the incoming messages are placed in an order, ar consumer processes also receive messages in their arrival sequence. By serializing requests, we mean that the requests (either for putting data or extracting data), which come to the server would be queued by the OS, and a single application thread will put them in the queue (we can assume that both kinds of requests, put and extract come to the same port) without any locking. It will be a possible lock-free solution, providing high throughput. This is a more viable solution because it can help us avoid the occurrence of race conditions.

Applications might use multiple queues with dedicated producers and consumers to keep the ordering cost per queue under check, although this comes at the cost of more complicated application logic.



Avoiding race conditions: Producers and consumers are serialized at both ends of the queue

In this lesson, we discussed some key considerations and challenges in the design process of a messaging queue and answered the following question

- Why is the order of messages important, and how do we enforce that order?

- How does ordering affect performance? How do we handle concurrency while accessing a queue?

Now, we are ready to start designing a distributed messaging queue.

← Back
Requirements of a Distributed Messaging Qu...

Next →
Design of a Distributed Messaging Queue: P...

✓ Mark as Completed

Introduction to Pub-sub

Learn about the use cases of the pub-sub system, how to define its requirements, and design the API for it.

We'll cover the following

Pub-sub messaging offers asynchronous communication. Let's explore the use cases where it is beneficial to have a pub-sub system.

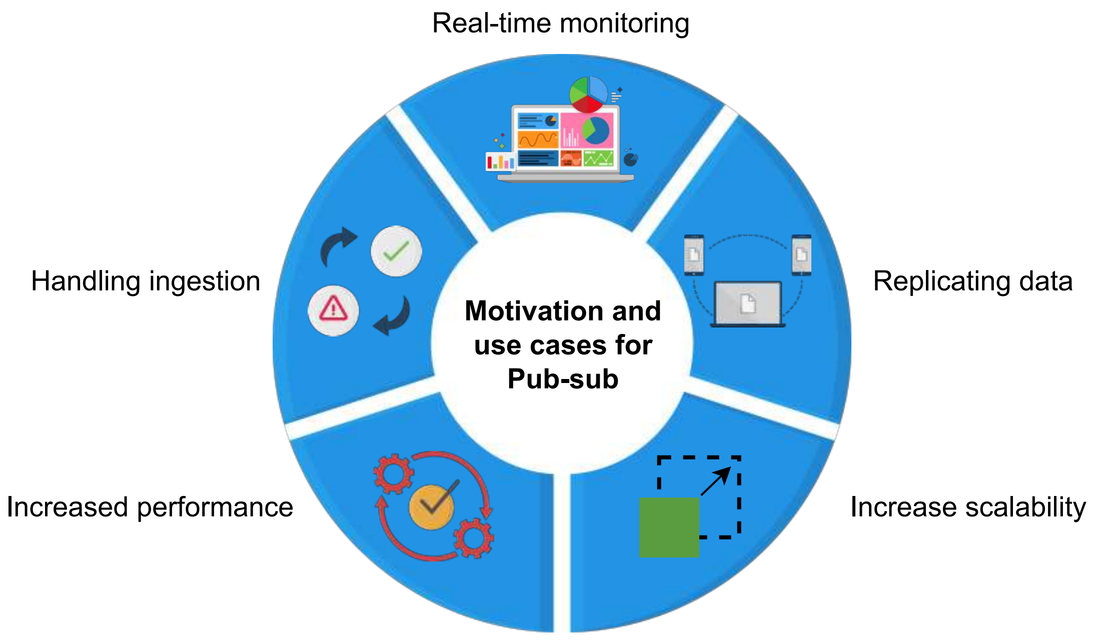
Use cases of pub-sub

A few use cases of pub-sub are listed below:

- **Improved performance**: The pub-sub system enables push-based distribution, alleviating the need for message recipients to check for new information and changes regularly. It encourages faster response times and lowers the delivery latency.
- **Handling ingestion**: The pub-sub helps in handling log ingestion. The user-interaction data can help us figure out useful analyses about the behavior of users. We can ingest a large amount of data to the pub-sul

system, so much so that it can deliver the data to any analytical system to understand the behavior patterns of users. Moreover, we can also log the details of the event that's happening while completing a request from the user. Large services like Meta use a pub-sub system called Scribe to know exactly who needs what data, and remove or archive processed or unwanted data. Doing this is necessary to manage an enormous amount of data.

- **Real-time monitoring**: Raw or processed messages of an application or system can be provided to multiple applications to monitor a system in real time.
- **Replicating data**: The pub-sub system can be used to distribute changes. For example, in a leader-follower protocol, the leader sends the changes to its followers via a pub-sub system. It allows followers to update their data asynchronously. The distributed caches can also refresh themselves by receiving the modifications asynchronously. Along the same lines, applications like WhatsApp that allow multiple views of the same conversation—for example, on a mobile phone and a computer's browser—can elegantly work using a pub-sub, where multiple views can act either as a publisher or a subscriber.



Motivation and use cases of the pub-sub system

Points to Ponder

Question 1

What are the similarities and differences between a pub-sub system and queues?

Show Answer

1 of 2

Requirements

We aim to design a pub-sub system that has the following requirements.

Functional requirements

Let's specify the functional requirements of a pub-sub system:

- **Create a topic**: The producer should be able to create a topic.
- **Write messages**: Producers should be able to write messages to the topic.
- **Subscription**: Consumers should be able to subscribe to the topic to
- **Read messages**: The consumer should be able to read messages from the topic.

- **Specify retention time**: The consumers should be able to specify the retention time after which the message should be deleted from the system.
- **Delete messages**: A message should be deleted from the topic or system after a certain retention period as defined by the user of the system.

Non-functional requirements

We consider the following non-functional requirements when designing a pub-sub system:

- **Scalable**: The system should scale with an increasing number of topics and increasing writing (by producers) and reading (by consumers) load.
- **Available**: The system should be highly available, so that producers can add their data and consumers can read data from it anytime.
- **Durability**: The system should be durable. Messages accepted from producers must not be lost and should be delivered to the intended subscribers.
- **Fault tolerance**: Our system should be able to operate in the event of failures.
- **Concurrent**: The system should handle concurrency issues where reading and writing are performed simultaneously.

API Design

We'll exclude some parameters from the functions below, such as the producer or consumer's identifier. Let's assume that this information is available from the underlying connection context. The API design for this problem is as follows:

Create a topic

The API call to create a topic should look like this:

```
create(topic_ID, topic_name)
```

This function returns an acknowledgment if it successfully creates a topic, or an error if it fails to do so.

Parameter	Description
`topic_ID`	It uniquely identifies the topic.
`topic_name`	It contains the name of the topic.

Write a message

The API call to write into the pub-sub system should look like this:

```
write(topic_ID, message)
```

The API call will write a `message` into a topic with an ID of `topic_ID`. Each message can have a maximum size of 1 MB. This function will return an acknowledgment if it successfully places the data in the systems, or an appropriate error if it fails.

Parameter	Description
`message`	The message to be written in the system.

Read a message

The API call to read data from the system should look like this:

```
read(topic_ID)
```

The topic is found using `topic_ID`, and the call will return an object containing the message to the caller.

Parameter	Description
`topic_ID`	It is the ID of the topic against which the message will be read.

Subscribe to a topic

The API call to subscribe to a topic from the system should look like this:

```
subscribe(topic_ID)
```

The function adds the consumer as a subscriber to the topic that has the `topic_ID`.

Parameter	Description
`topic_ID`	The ID of the topic to which the consumer will be subscribed.

Unsubscribe from a topic

The API call to unsubscribe from a topic from the system should look like this:

```
unsubscribe(topic_ID)
```

The function removes the consumer as a subscriber from the topic that has the `topic_ID`.

Parameter	Description
`topic_ID`	The ID of the topic against which the consumers will be unsubscribed.

Delete a topic

The API call to delete a topic from the system should look like this:

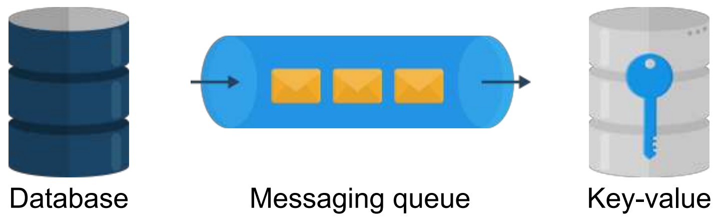
```
delete_topic(topic_ID)
```

The function deletes the topic on the basis of the `topic_ID`.

Parameter	Description
`topic_ID`	The ID of the topic which is to be deleted.

Building blocks we will use

The design of pub-sub utilizes many building blocks that have been discussed in the initial chapters. We'll consider the following lessons on building blocks.



The building blocks we'll use

- **Database**: We'll use databases to store information like subscription details.
- **Distributed messaging queue**: We'll use use a messaging queue to store messages sent by the producer.
- **Key-value**: We'll use a key-value store to hold information about consumers.

In the next lesson, we'll focus on designing a pub-sub system.

Back

System Design: The Pub-sub Abstraction

Next

Design of a Pub-sub System

✓ Mark as Completed

Design of a Pub-sub System

Dive into designing a pub-sub system and its components.

We'll cover the following

First design

In the previous lesson, we discussed that a producer writes into topics, and consumers subscribe to a topic to read messages from that topic. Since new messages are added at the end of the queue, we can use distributed messaging queues for topics.

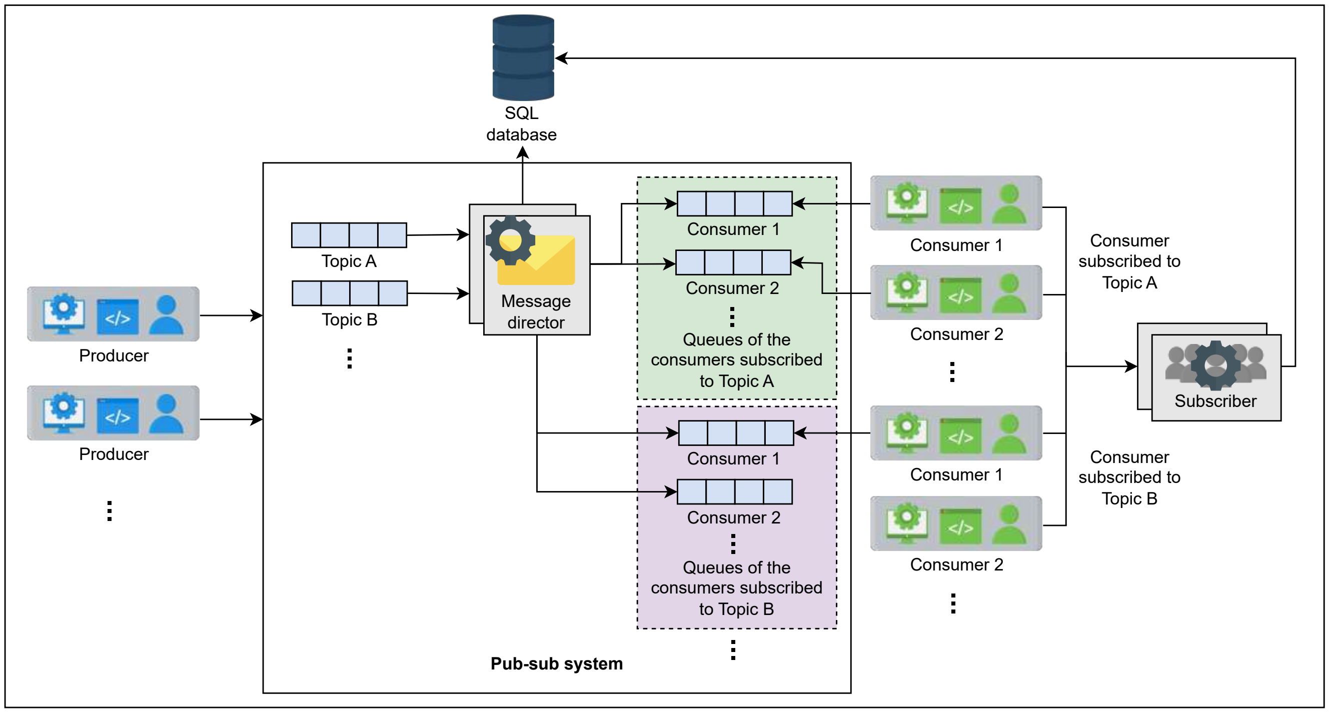
The components we'll need have been listed below:

- **Topic queue**: Each topic will be a distributed messaging queue so we can store the messages sent to us from the producer. A producer will write their messages to that queue.

- **Database**: We'll use a relational database that will store the subscription details. For example, we need to store which consumer has subscribed to which topic so we can provide the consumers with their desired messages. We'll use a relational database since our consumer-related data is structured and we want to ensure our data integrity.
- **Message director**: This service will read the message from the topic queue, fetch the consumers from the database, and send the message to the consumer queue.
- **Consumer queue**: The message from the topic queue will be copied to the consumer's queue so the consumer can read the message. For each consumer, we'll define a separate distributed queue.
- **Subscriber**: When the consumer requests a subscription to a topic, this service will add an entry into the database.

The consumer will subscribe to a topic, and the system will add the subscriber's details to the database. The producer will write into the topics, and the message director will read the message from the queue, fetch the details to whom it should add the message, and send it to them. The consumers will consume the message from their queue.

> **Note**: We'll use fail-over services for the message director and subscriber to guard against failures.



Using the distributed messaging queue

Using the distributed messaging queues makes our design simple. However, the huge number of queues needed is a significant concern. If we have millions of subscribers for thousands of topics, defining and maintaining millions of queues is expensive. Moreover, we'll copy the same message for a topic in all subscriber queues, which is unnecessary duplication and takes up space.

Points to Ponder

Question 1

Is there a way to avoid maintaining a separate queue for each reader?

Show Answer

1 of 2

Second design

Let’s consider another approach to designing a pub-sub system.

High-level design

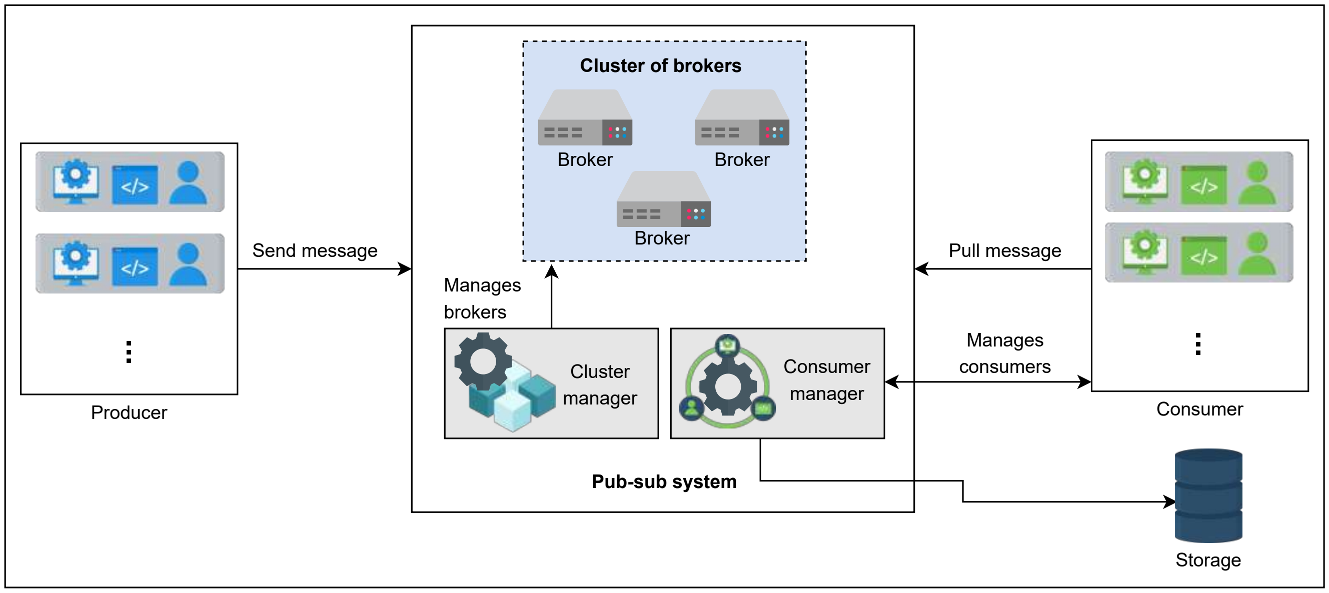
At a high level, the pub-sub system will have the following components:

- **Broker**: This server will handle the messages. It will store the messages sent from the producer and allow the consumers to read them.
- **Cluster manager**: We’ll have numerous broker servers to cater to our scalability needs. We need a cluster manager to supervise the broker’s health. It will notify us if a broker fails.
- **Storage**: We’ll use a relational database to store consumer details, such as subscription information and retention period.
- **Consumer manager**: This is responsible for managing the consumers. For example, it will verify if the consumer is authorized to read a message from a certain topic or not.

Besides these components, we also have the following design considerations:

- **Acknowledgment**: An acknowledgment is used to notify the producer that the received message has been stored successfully. The system will wait for an acknowledgment from the consumer if it has successfully consumed the message.
- **Retention time**: The consumers can specify the retention period time of their messages. The default will be seven days, but it is configurable. Some applications like banking applications require the data to be

stored for a few weeks as a business requirement, while an analytical application might not need the data after consumption.

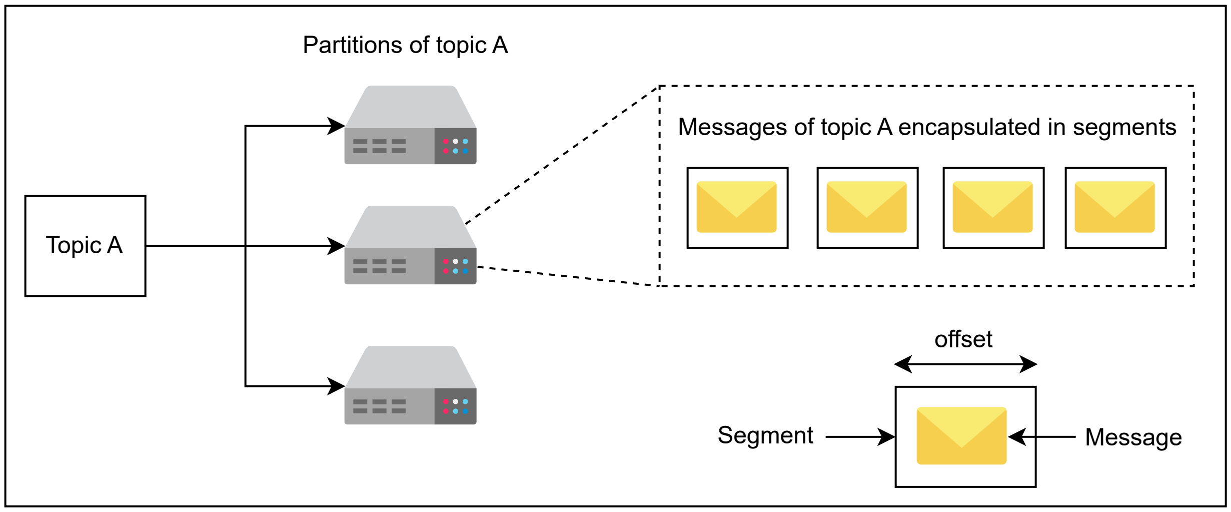


High-level design of a pub-sub system

Let’s understand the role of each component in detail.

Broker

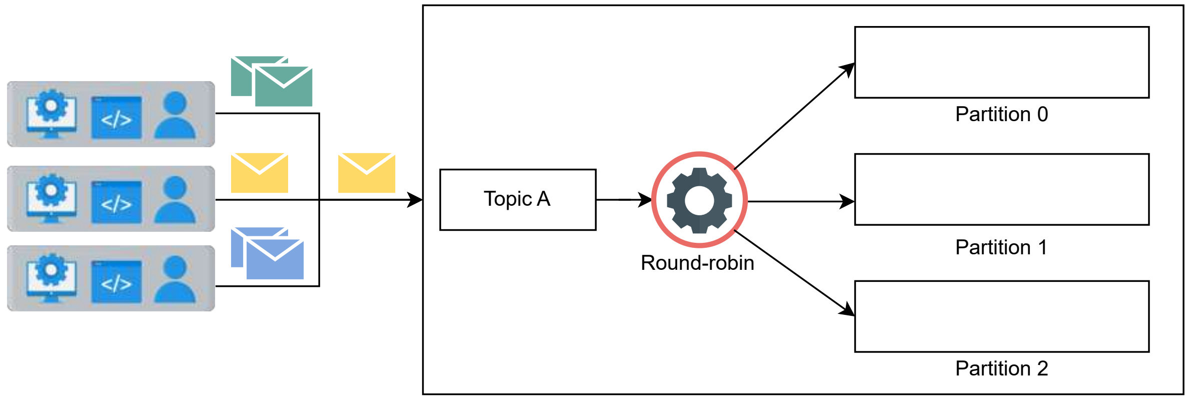
The broker server is the core component of our pub-sub system. It will handle write and read requests. A broker will have multiple topics where each topic can have multiple **partitions** associated with it. We use partitions to store messages in the local storage for persistence. Consequently, this improves availability. Partitions contain messages encapsulated in **segments**. Segments help identify the start and end of a message using an offset address. Using segments, consumers consume the message of their choice from a partition by reading from a specific offset address. The illustration below depicts the concept that has been described above.



A depiction of how messages are stored within segments inside a partition

As we know, a **topic** is a persistent sequence of messages stored in the local storage of the broker. Once the data has been added to the topic, it cannot be modified. Reading and writing a message from or to a topic is an I/O task for

the topics into multiple partitions. The data belonging to a single topic can be present in numerous partitions. For example, let's assume have Topic A and we allocate three partitions for it. The producers will send their messages to the relevant topic. The messages received will be sent to various partitions on basis of the round-robin algorithm. We'll use a variation of round-robin: weighted round-robin. The following slides show how the messages are stored in various partitions belonging to a single topic.



Producers create messages of the same topic and send them to the system

1 of 9

Points to Ponder

Question 1

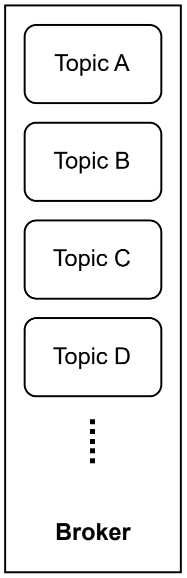
Strict ordering ensures that the messages are stored in the order in which they are produced. How can we ensure strict ordering for our messages?

Show Answer

1 of 4

We'll allocate the partitions to various brokers in the system. This just means that different partitions of the same topic will be in different brokers. We'll follow strict ordering in partitions by adding newer content at the end of existing messages.

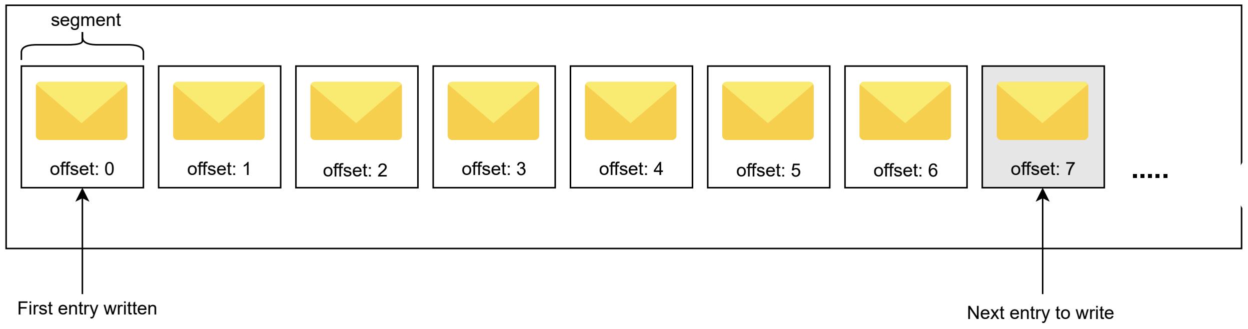
Consider the slides below. We have various brokers in our system. Each broker has different topics. The topic is divided into multiple partitions.



A broker contains multiple topics

1 of 3

We discussed that a message will be stored in a segment. We'll identify each segment using an offset. Since these are immutable records, the readers are independent and they can read messages anywhere from this file using the necessary API functions. The following slides show the segment level detail.



A new entry will be added at the end of the file

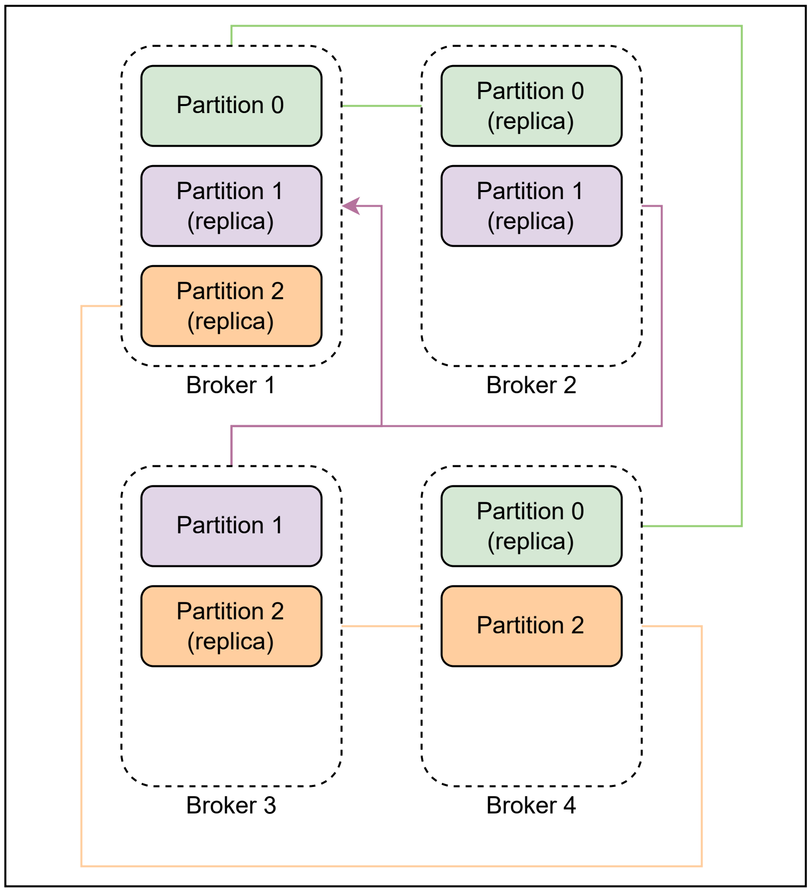
1 of 2

The broker solved the problems that our first design had. We avoided a large number of queues by partitioning the topic. We introduced parallelism using partitions that avoided bottlenecks while consuming the message.

Cluster manager

We'll have multiple brokers in our cluster. The cluster manager will perform the following tasks:

- **Broker and topics registry**: This stores the list of topics for each broker.
- **Manage replication**: The cluster manager manages replication by using the leader-follower approach. One of the brokers is the leader. If it fails, the manager decides who the next leader is. In case the follower fails, it adds a new broker and makes sure to turn it into an updated follower. It updates the metadata accordingly. We'll keep three replicas of each partition on different brokers.



Replication at the partitioning level

Consumer manager

The consumer manager will manage the consumers. It has the following responsibilities:

- **Verify the consumer**: The manager will fetch the data from the database and verify if the consumer is allowed to read a certain message. For example, if the consumer has subscribed to Topic A (but not to Topic B), then it should not be allowed to read from Topic B. The consumer manager verifies the consumer's request.
- **Retention time management**: The manager will also verify if the consumer is allowed to read the specific message or not. If, according to its retention time, the message should be inaccessible to the consumer, then it will not allow the consumer to read the message.

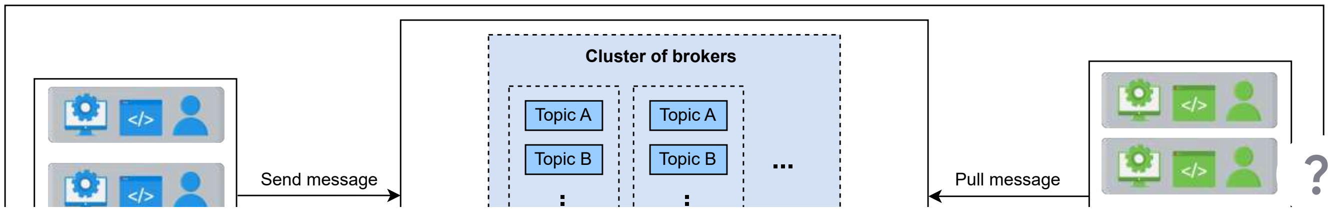
- **Message receiving options management**: There are two methods for consumers to get data. The first is that our system pushes the data to its consumers. This method may result in overloading the consumers with continuous messages. Another approach is for consumers to request the system to read data from a specific topic. The drawback is that a few consumers might want to know about a message as soon as it is published, but we do not support this function.

  Therefore, we'll support both techniques. Each consumer will inform the broker that it wants the data to be pushed automatically or it needs the data to read itself. We can avoid overloading the consumer and also provide liberty to the consumer. We'll save this information in the relational database along with other consumer details.

- **Allow multiple reads**: The consumer manager stores the offset information of each consumer. We'll use a key-value to store offset information against each consumer. It allows fast fetching and increases the availability of the consumers. If Consumer 1 has read from offset 0 and has sent the acknowledgment, we'll store it. So, when the consumer wants to read again, we can provide the next offset to the reader for reading the message.

Finalized design

The finalized design of our pub-sub system is shown below.



Requirements of a Rate Limiter’s Design

Understand the requirements and important concepts of a rate limiter.

We'll cover the following

Requirements

Our focus in this lesson is to design a rate limiter with the following functional and non-functional requirements.

Functional requirements

- To limit the number of requests a client can send to an API within a time window.
- To make the limit of requests per window configurable.
- To make sure that the client gets a message (error or notification) whenever the defined threshold is crossed within a single server or combination of servers.

Non-functional requirements

- **Availability:** Essentially, the rate limiter protects our system. Therefore, it should be highly available.
- **Low latency:** Because all API requests pass through the rate limiter, it should work with a minimum latency without affecting the user experience.
- **Scalability:** Our design should be highly scalable. It should be able to rate limit an increasing number of clients’ requests over time.

Types of throttling

A rate limiter can perform three types of throttling.

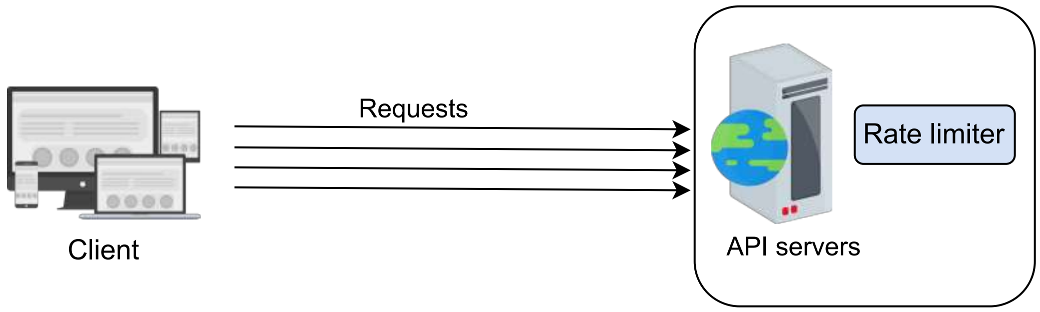
1. **Hard throttling:** This type of throttling puts a hard limit on the number of API requests. So, whenever a request exceeds the limit, it is discarded.
2. **Soft throttling:** Under soft throttling, the number of requests can exceed the predefined limit by a certain percentage. For example, if our system has a predefined limit of 500 messages per minute with a 5% exceed in the limit, we can let the client send 525 requests per minute.
3. **Elastic or dynamic throttling:** In this throttling, the number of requests can cross the predefined limit if the system has excess resources available. However, there is no specific percentage defined for the upper limit. For example, if our system allows 500 requests per minute, it can let the user send more than 500 requests when free resources are available.

Throttling at the Operating System (OS) level

Where to place the rate limiter

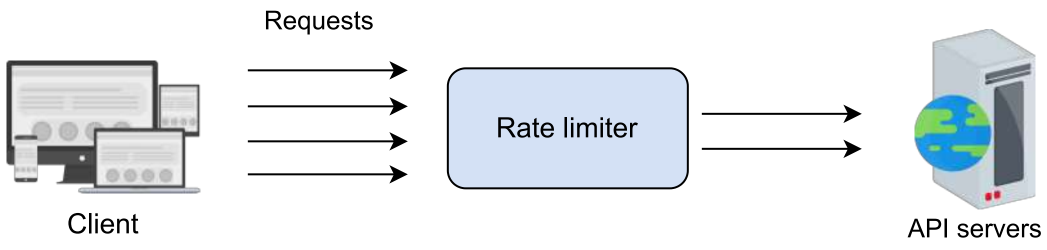
There are three different ways to place the rate limiter.

1. **On the client side:** It is easy to place the rate limiter on the client side. However, this strategy is not safe because it can easily be tampered with by malicious activity. Moreover, the configuration on the client side is also difficult to apply in this approach.
2. **On the server side:** As shown in the following figure, the rate limiter is placed on the server-side. In this approach, a server receives a request that is passed through the rate limiter that resides on the server.



Rate limiter placed at the server side

3. **As middleware:** In this strategy, the rate limiter acts as middleware, throttling requests to API servers as shown in the following figure.



Rate limiter as middleware

Placing a rate limiter is dependent on a number of factors and is a subjective decision, based on the organization's technology stack, engineering resources, priorities, plan, goals, and so on.

Note: Many modern services use APIs to provide their functionality to the clients. API endpoints can be a good vantage point to rate limit

> the incoming client traffic because all traffic passes through them.

Two models for implementing a rate limiter

One rate limiter might not be enough to handle enormous traffic to support millions of users. Therefore, a better option is to use multiple rate limiters as a cluster of independent nodes. Since there will be numerous rate limiters with their corresponding counters (or their rate limit), there are two ways to use databases to store, retrieve, and update the counters along with the user information.

1. **A rate limiter with a centralized database:** In this approach, rate limiters interact with a centralized database, preferably Redis or Cassandra. The advantage of this model is that the counters are stored in centralized databases. Therefore, a client can’t exceed the predefined limit. However, there are a few drawbacks to this approach. It causes an increase in latency if an enormous number of requests hit the centralized database. Another extensive problem is the potential for race conditions in highly concurrent requests (or associated lock contention).
2. **A rate limiter with a distributed database:** Using an independent cluster of nodes is another approach where the rate-limiting state is in a distributed database. In this approach, each node has to track the rate limit. The problem with this approach is that a client could exceed a rate limit—at least momentarily, while the state is being collected from everyone—when sending requests to different nodes (rate-limiters). To enforce the limit, we must set up sticky sessions in the load balancer t send each consumer to exactly one node. However, this approach lacks fault tolerance and poses scaling problems when the nodes get overloaded.

Aside from the above two concepts, another problem is whether to use a global counter shared by all the incoming requests or individual counters per user. For example, the token bucket algorithm can be implemented in two ways. In the first method, all requests can share the total number of tokens in a single bucket, while in the second method, individual buckets are assigned to users. The choice of using shared or separate counters (or buckets) depends on the use case and the rate-limiting rules.

Points to Ponder

Question 1

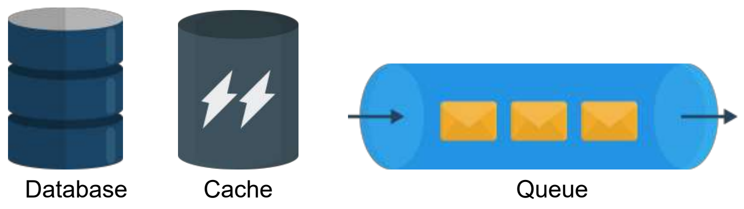
Can a rate limiter be used as a load balancer?

Show Answer

1 of 2

Building blocks we will use

The design of the rate limiter utilizes the following building blocks that we discussed in the initial chapters.



Building blocks in the design of a rate limiter

- **Databases** are used to store rules defined by a service provider and metadata of users using the service.
- **Caches** are used to cache the rules and users’ data for frequent access.
- **Queues** are essential for holding the incoming requests that are allowed by the rate limiter.

In the next lesson, we’ll focus on a high-level and detailed design of a rate limiter based on the requirements discussed in this lesson.

← Back
System Design: The Rate Limiter

Next →
Design of a Rate Limiter

✓ Mark as Completed

Design of a Rate Limiter

Learn to design rate limiters that help gauge and throttle resources being used across our system.

We'll cover the following

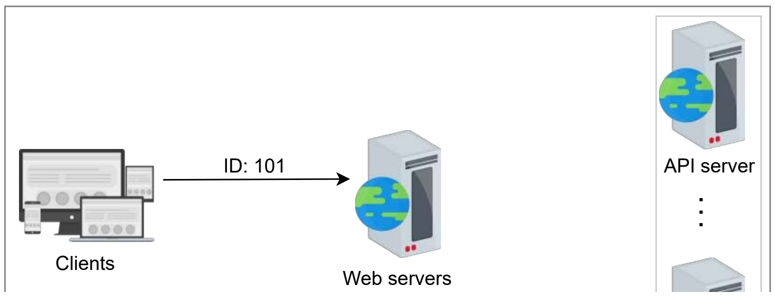
High-level design

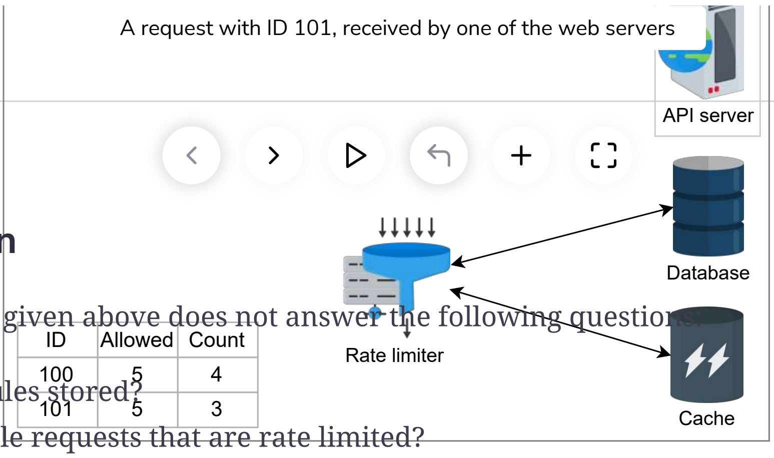
A rate limiter can be deployed as a separate service that will interact with a web server, as shown in the figure below. When a request is received, the rate limiter suggests whether the request should be forwarded to the server or not. The rate limiter consists of rules that should be followed by each incoming request. These rules define the throttling limit for each operation. Let's go through a rate limiter rule from Lyft, which has open-sourced its rate limiting component.

```
domain: messaging
descriptors:
   -key: message_type
    value: marketing
    rate_limit:
               unit: day
               request_per_unit: 5
```

Rate-limiting rules from Lyft

In the above rate-limiting rule, the `unit` is set to `day` and the `request_per_unit` is set to `5`. These parameters define that the system can allow five marketing messages per day.





ID	Allowed	Count
100	5	4
101	5	3

A request with ID 101, received by one of the web servers

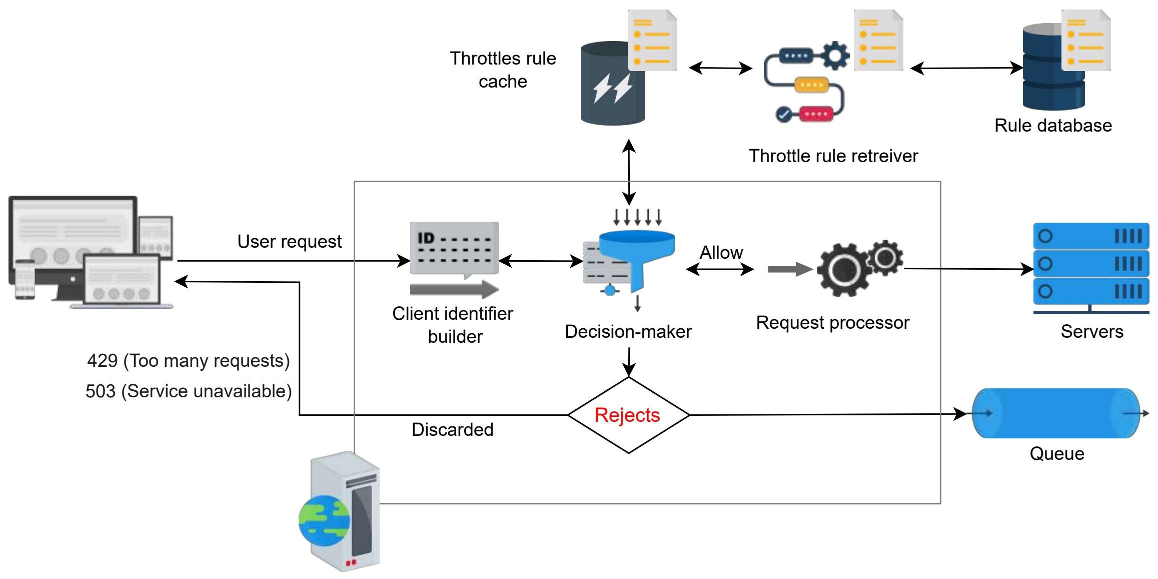
1 of 6

Detailed design

The high-level design given above does not answer the following questions:

- Where are the rules stored?
- How do we handle requests that are rate limited?

In this section, we'll first expand the high-level architecture into several other essential components. We'll also explain each component in detail, as shown in the following figure.



The rate limiter accepts or rejects requests based on throttle rules

Let's discuss each component that is present in the detailed design of a rate limiter.

Rule database: This is the database, consisting of rules defined by the service owner. Each rule specifies the number of requests allowed for a particular client per unit of time.

Rules retriever: This is a background process that periodically checks for any modifications to the rules in the database. The rule cache is updated if there are any modifications made to the existing rules.

Throttle rules cache: The cache consists of rules retrieved from the **rule database**. The cache serves a rate-limiter request faster than persistent storage. As a result, it increases the performance of the system. So, when the rate limiter receives a request against an ID (key), it checks the ID against the rules in the cache.

Decision-maker: This component is responsible for making decisions against the rules in the cache. This component works based on one of the rate-limiting algorithms that are discussed in the next lesson.

Client identifier builder: This component generates a unique ID for a request received from a client. This could be a remote IP address, login ID, or a combination of several other attributes, due to which a sequencer can't be used here. This ID is considered as a key to store the user data in the key-value database. So, this key is passed to the **decision-maker** for further service decisions.

In case the predefined limit is crossed, APIs return an HTTP response code `429 Too Many Requests`, and one of the following strategies is applied to the request:

- Drop the request and return a specific response to the client, such as "too many requests" or "service unavailable."
- If some requests are rate limited due to a system overload, we can keep those requests in a queue to be processed later.

Request processing

When a request is received, the *client identifier builder* identifies the request and forwards it to the *decision-maker*. The decision-maker determines the services required by request, then checks the cache against the number of requests allowed, as well as the rules provided by the service owner. If the request does not exceed the count limit, it is forwarded to the *request processor*, which is responsible for serving the request.

The decision-maker takes decisions based on the throttling algorithms. The throttling can be hard, soft, or elastic. Based on **soft or elastic throttling**, requests are allowed more than the defined limit. These requests are either served or kept in the queue and served later, upon the availability of resources. Similarly, if **hard throttling** is used, requests are rejected, and a response error is sent back to the client.

Point to Ponder

Question

In the event of a failure, a rate limiter is unable to perform the task of throttling. In these scenarios, should the request be accepted or rejected?

Show Answer

Race condition

There is a possibility of a race condition in a situation of high concurrency request patterns. It happens when the “get-then-set” approach is followed, wherein the current counter is retrieved, incremented, and then pushed back to the database. While following this approach, some additional requests can come through that could leave the incremented counter invalid. This allows a client to send a very high rate of requests, bypassing the rate-limiting controls. To avoid this problem, the locking mechanism can be used, where one process can update the counter at a time while others wait for the lock to be released. Since this approach can cause a potential bottleneck, it significantly degrades performance and does not scale well.

Another method that could be used is the “set-then-get” approach, wherein a value is incremented in a very performant fashion, avoiding the locking approach. This approach works if there’s minimum contention. However, one might use other approaches where the allowed quota is divided into multiple places and divide the load on them, or use sharded counters to scale an approach.

> **Note:** We can use sharded counters for rate-limiting under the highly concurrent nature of traffic. By increasing the number of shards, we reduce write contention. Since we have to collect counters from all shards, our reading may slow down.

A rate limiter should not be on the client’s critical path

Let’s assume a real-time scenario where millions of requests hit the front-end servers. Each request will retrieve, update, and push back the count to the respective cache. After all these operations, the request is sent forward to be served. This approach could cause latency if there is a high number of requests. To avoid numerous computations in the client’s critical path, we should divide the work into offline and online parts.

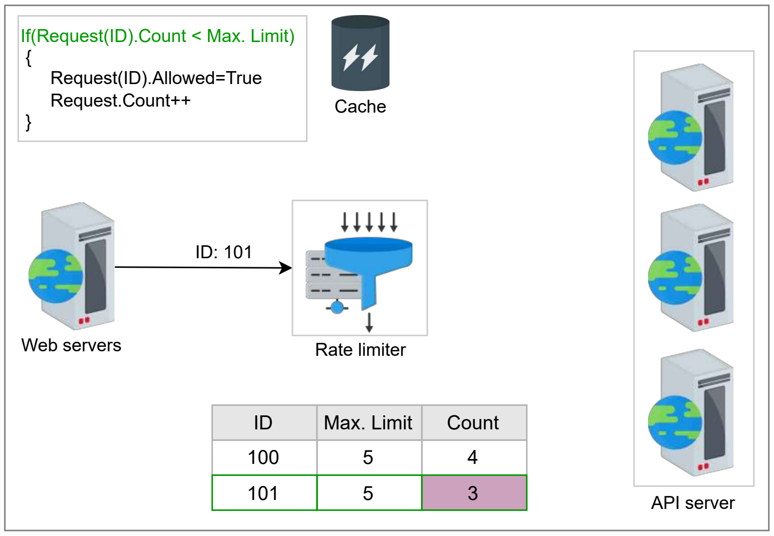
Initially, when a client’s request is received, the system will just check the respective count. If it is less than the maximum limit, the system will allow the client’s request. In the second phase, the system updates the respective count and cache offline. For a few requests, this won’t have any effect on the performance, but for millions of requests, this approach increases performance significantly.

Let’s understand the online and offline updates approach with an example. In the following set of illustrations, when a request is received, its ID is forwarded to the rate limiter that will check the condition `if(request(ID).Count <= Max. Limit` by retrieving data from the cache. For simplicity, assume that one of the requests ID is 101, that is, `request(ID) = 101`. The following table shows the number of requests made by each client and the maximum number of requests per unit time that a client can make.

Request ID	Maximum Limit	Count
100	5	4

101	5	3

If the condition is true, the rate limiter will first respond back to the front-end server with an `Allowed` signal. The corresponding `count` and other relevant information are updated offline in the next steps. The rate limiter writes back the updated data in the cache. Following this approach reduces latency and avoids the contention that incoming requests could have caused.



ID	Max. Limit	Count
100	5	4
101	5	3

A request with ID:101 is received. The count for this is 3

1 of 4

> **Note:** We've seen a form of rate limiting in TCP network protocol, where the recipient can throttle the sender by advertising the size of the window (the outstanding data a recipient is willing to receive). The sender sends the minimum value of either the congestion window or the advertised window. Many network traffic shapers use similar mechanisms to provide preferential treatment to different network flows.

Conclusion

In this lesson, we discussed the design of a rate limiter in distributed systems. Let's analyze the non-functional requirements we promised in the previous lesson.

- **Availability:** If a rate limiter fails, multiple rate limiters will be available to handle the incoming requests. So, a single point of failure is eliminated.

- **Low latency:** Our system retrieves and updates the data of each incoming request from the cache instead of the database. First, the incoming requests are forwarded if they do not exceed the rate limit, and then the cache and database are updated.
- **Scalability:** The number of rate limiters can be increased or decreased based on the number of incoming requests within the defined limit.

Now, our system provides high availability, low latency, and scalability in light of the above discussion.

← Back
Requirements of a Rate Limiter's Design

Next →
Rate Limiter Algorithms

✓ Mark as Completed

Rate Limiter Algorithms

Understand the working of various rate limiter algorithms.

We'll cover the following

Algorithms for rate limiting

The task of a rate limiter is directed by highly efficient algorithms, each of which has distinct advantages and disadvantages. However, there is always a choice to choose an algorithm or combination of algorithms depending on what we need at a given time. While different algorithms are used in addition to those below, we'll take a look at the following popular algorithms.

- Token bucket
- Leaking bucket
- Fixed window counter
- Sliding window log
- Sliding window counter

This algorithm uses the analogy of a bucket with a predefined capacity of tokens. The bucket is periodically filled with tokens at a constant rate. A token can be considered as a packet of some specific size. Hence, the algorithm checks for the token in the bucket each time we receive a request. There should be at least one token to process the request further.

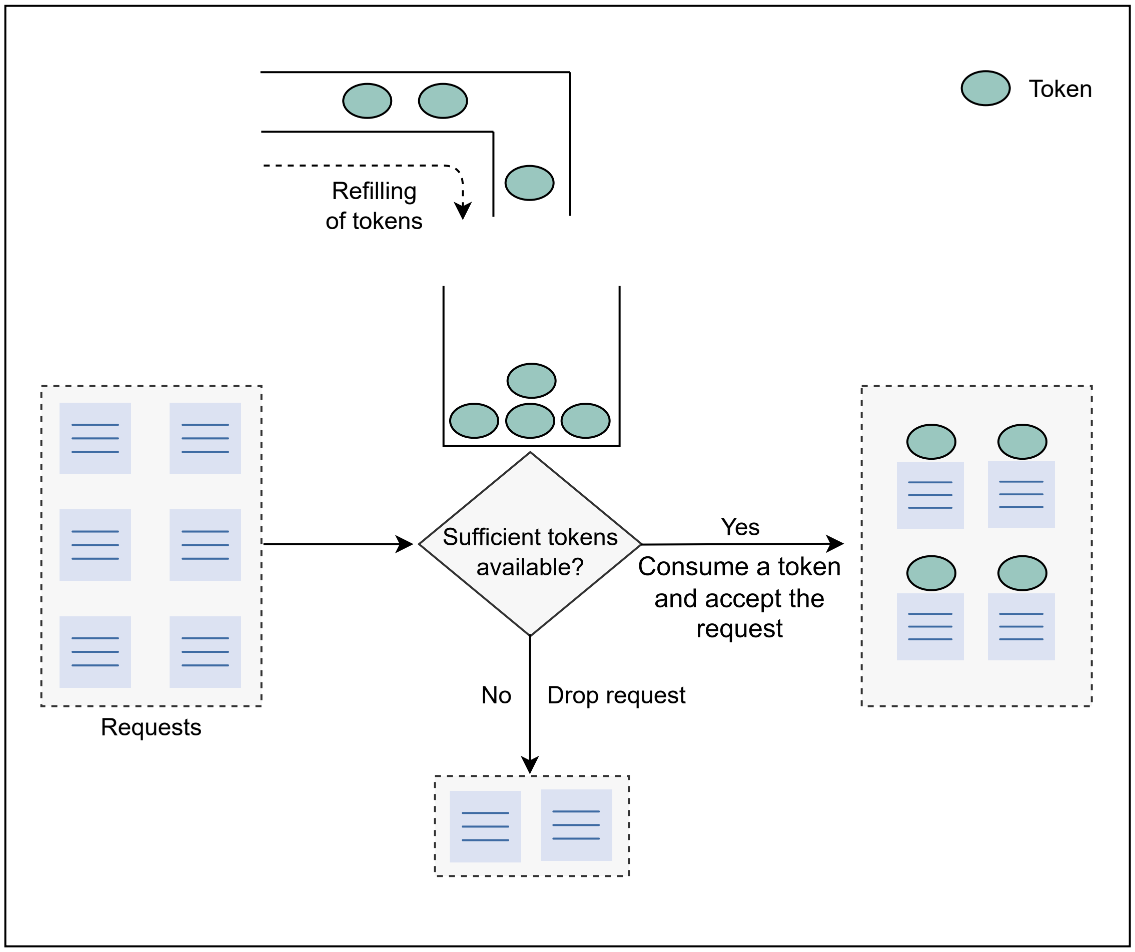
The flow of the token bucket algorithm is as follows:

Assume that we have a predefined rate limit of R and the total capacity of the bucket is C.

1. The algorithm adds a new token to the bucket after every $\frac{1}{R}$ seconds.

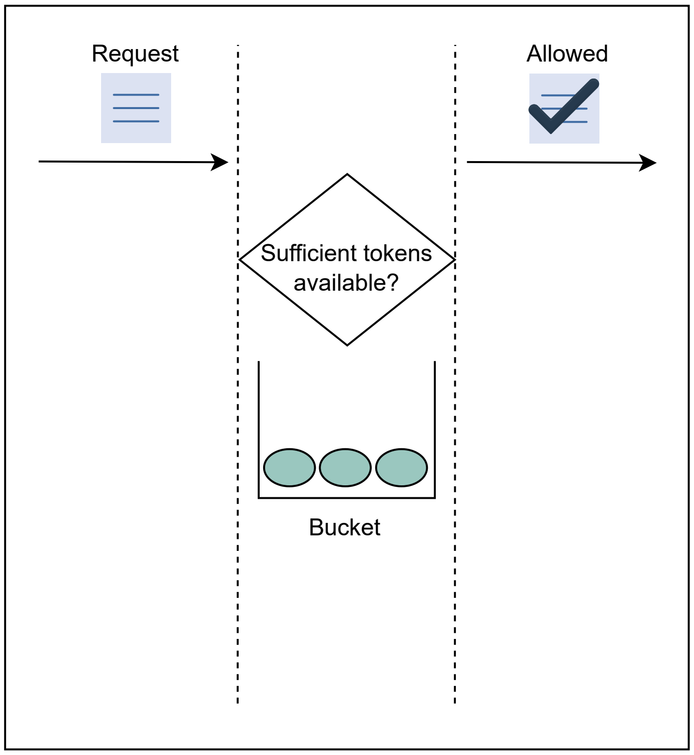
2. The algorithm discards the new incoming tokens when the number of tokens in the bucket is equal to the total capacity C of the bucket.
3. If there are N incoming requests and the bucket has at least N tokens, the tokens are consumed, and requests are forwarded for further processing.
4. If there are N incoming requests and the bucket has a lower number of tokens, then the number of requests accepted equals the number of available tokens in the bucket.

The following illustration represents the working of the token bucket algorithm.


Token
Refilling of tokens
Requests
Sufficient tokens available?
Yes
Consume a token and accept the request
No
Drop request

How the token bucket algorithm works

The following illustration demonstrates how token consumption and rate-limiting logic work. In this example, the capacity of the bucket is three, and it is refilled at a rate of three tokens per minute.



Initially, there are three tokens in the bucket. A request arrives within a minute and consumes a token from the bucket

1 of 4

Essential parameters

We require the following essential parameters to implement the token bucket algorithm:

- **Bucket capacity** (C)**:** The maximum number of tokens that can reside in the bucket.
- **Rate limit** (R)**:** The number of requests we want to limit per unit tim
- **Refill rate** $(\frac{1}{R})$**:** The number of tokens put into the bucket per unit time.

- **Requests count (N):** This parameter tracks the number of incoming requests and compares them with the bucket's capacity.

Advantages

- This algorithm can cause a burst of traffic as long as there are enough tokens in the bucket.
- It is space efficient. The memory needed for the algorithm is nominal due to limited states.

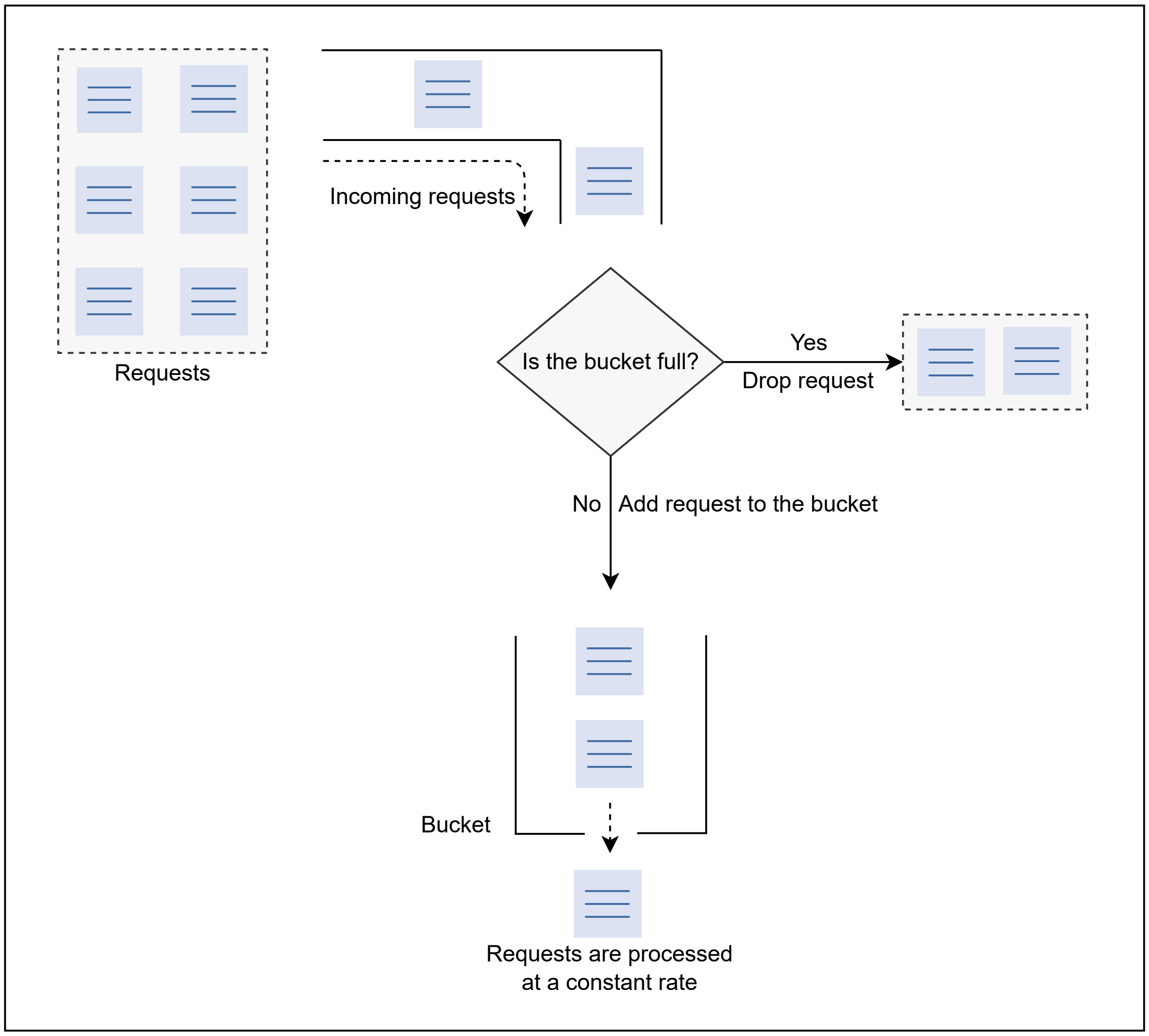
Disadvantages

- From the implementation perspective, a lock might require taking a token from the bucket that can increase the request's processing delay if contention on the lock increases.
- Choosing an optimal value for the essential parameters is a difficult task.

The leaking bucket algorithm

The **leaking bucket algorithm** is a variant of the token bucket algorithm with slight modifications. Instead of using tokens, the leaking bucket algorithm uses a bucket to contain incoming requests and processes them at a constant outgoing rate. This algorithm uses the analogy of a water bucket leaking at a constant rate. Similarly, in this algorithm, the requests arrive at a variable rate. The algorithm process these requests at a constant rate in a first-in-first-out (FIFO) order.

Let's look at how the leaking bucket algorithm works in the illustration below:



How the leaking bucket algorithm works

Essential parameters

The leaking bucket algorithm requires the following parameters.

- **Bucket capacity** (C)**:** This determines the maximum capacity of the bucket. The algorithm will discard the incoming requests when the bucket reached its maximum limit of C.
- **Inflow rate** (R_{in})**:** This parameter shows the inflow rate of requests. This is a varying quantity that depends on the application and nature of requests. We use this parameter to find the initial capacity of the bucket.

- **Outflow rate (R_{out}):** This determines the number of requests processed per unit time.

Advantages

- Due to a constant outflow rate (R_{out}), it avoids the burst of requests, unlike the token bucket algorithm.
- This algorithm is also space efficient since it requires just three states: inflow rate (R_{in}), outflow rate (R_{out}), and bucket capacity (C).
- Since requests are processed at a fixed rate, it is suitable for applications with a stable outflow rate.

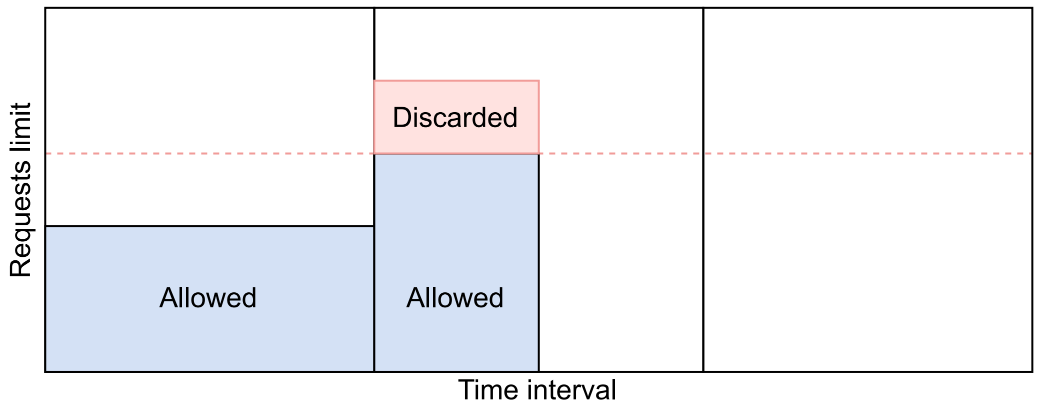
Disadvantages

- A burst of requests can fill the bucket, and if not processed in the specified time, recent requests can take a hit.
- Determining an optimal bucket size and outflow rate is a challenge.

Fixed window counter algorithm

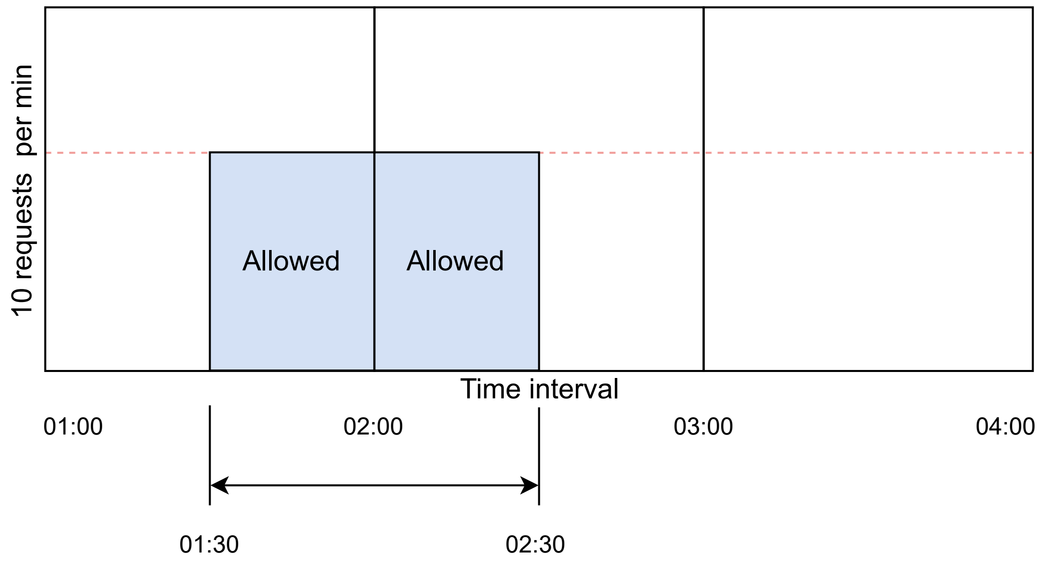
This algorithm divides the time into fixed intervals called **windows** and assigns a counter to each window. When a specific window receives a request, the counter is incremented by one. Once the counter reaches its limit, new requests are discarded in that window.

As shown in the below figure, a dotted line represents the limit in each window. If the counter is lower than the limit, forward the request; otherwise, discard the request.



Fixed window counter algorithm: Discard the request exceeding the limit

There is a significant problem with this algorithm. A burst of traffic greater than the allowed requests can occur at the edges of the window. In the below figure, the system allows a maximum of ten requests per minute. However, the number of requests in the one-minute window from 01:30 to 02:30 is 20, which is greater than the allowed number of requests.



Edge case problem in the fixed window counter algorithm. The number of requests in one minute from 01:30 to 02:30 exceeds the predefined limit of 10 requests per minute

Essential parameters

The fixed window counter algorithm requires the following parameters:

- **Window size (W):** It represents the size of the time window. It can be a minute, an hour, or any other suitable time slice.
- **Rate limit (R):** It shows the number of requests allowed per time window.
- **Requests count (N):** This parameter shows the number of incoming requests per window. The incoming requests are allowed if N is less than or equal to R.

Advantages

- It is also space efficient due to constraints on the rate of requests.
- As compared to token bucket-style algorithms (that discard the new requests if there aren't enough tokens), this algorithm services the new requests.

Disadvantages

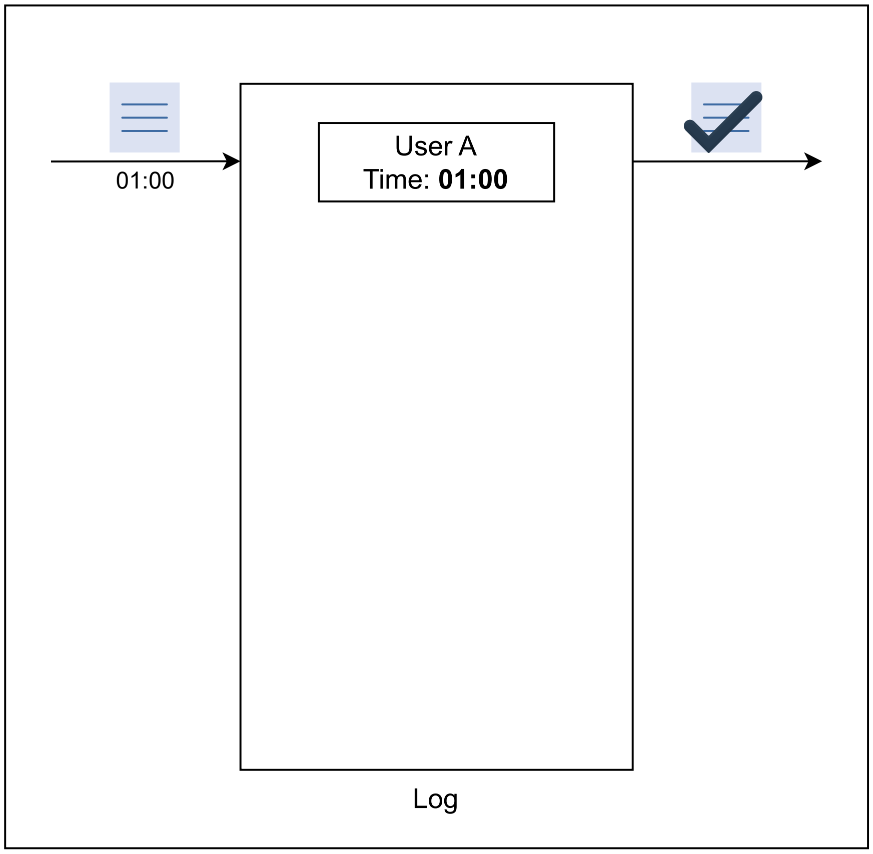
- A consistent burst of traffic (twice the number of allowed requests per window) at the window edges could cause a potential decrease in performance.

Sliding window log algorithm

The **sliding window log algorithm** keeps track of each incoming request. When a request arrives, its arrival time is stored in a hash map, usually known as the log. The logs are sorted based on the time stamps of incoming requests. The requests are allowed depending on the size of the log and arrival time.

The main advantage of this algorithm is that it doesn't suffer from the edge conditions, as compared to the **fixed window counter** algorithm.

Let's understand how the sliding window log algorithm works in the illustration below. Assume that we have a maximum rate limit of two requests in a minute.



A new request arrives at 01:00. Its arrival time is added to the log and the request is accepted. The time window is marked from 01:00 to 02:00

1 of 4

Essential parameters

The following parameters are required to implement the sliding window lo
algorithm:

- **Log size (L):** This parameter is similar to the rate limit (R) as it determines the number of requests allowed in a specific time frame.

- **Arrival time** (T)**:** This parameter tracks incoming requests' time stamps and determines their count.
- **Time range** (T_r)**:** This parameter determines the time frame. The time stamps of the old requests are deleted if they do not fall in this range. The start time of the window is defined based on the first incoming request and expires after one minute. Similarly, when another request after the expiry time arrives the window ranges are updated accordingly.

Advantages

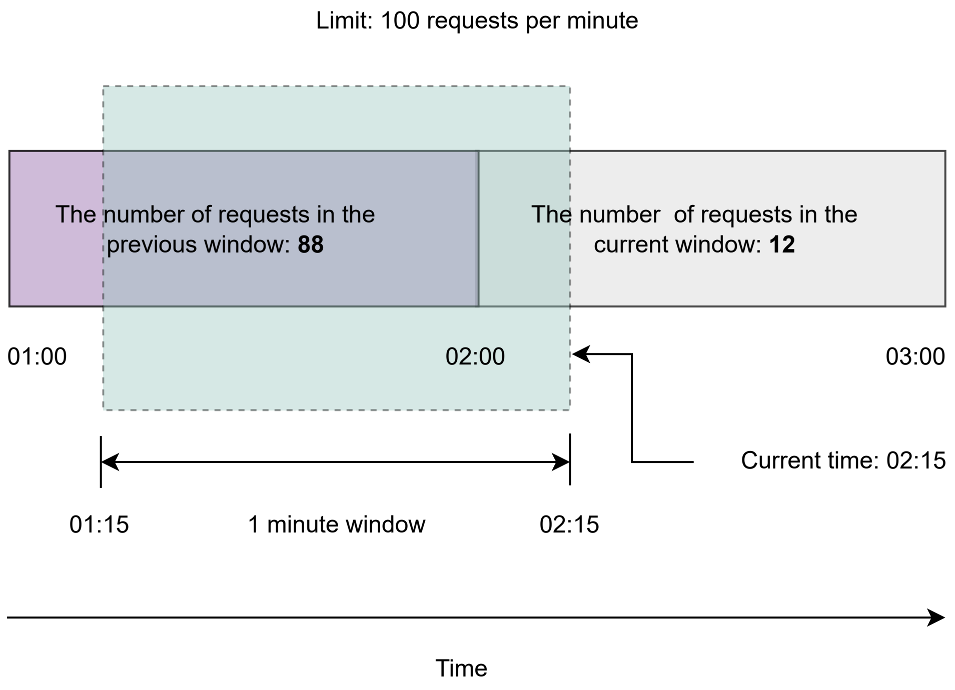
- The algorithm doesn't suffer from the boundary conditions of fixed windows.

Disadvantages

- It consumes extra memory for storing additional information, the time stamps of incoming requests. It keeps the time stamps to provide a dynamic window, even if the request is rejected.

Sliding window counter algorithm

Unlike the previously fixed window algorithm, the **sliding window counter algorithm** doesn't limit the requests based on fixed time units. This algorithm takes into account both the fixed window counter and sliding window log algorithms to make the flow of requests more smooth. Let's look at the flow of the algorithm in the below figure.


Limit: 100 requests per minute
The number of requests in the previous window: 88
The number of requests in the current window: 12
01:00
02:00
03:00
Current time: 02:15
01:15
1 minute window
02:15
Time

A sliding window counter algorithm, where the green shaded area shows the rolling window of 1 minute

In the above figure, we've 88 requests in the previous window while 12 in the current window. We've set the rate limit to 100 requests per minute. Further, the rolling window overlaps 15 seconds with the current window. Now assume that a new request arrives at 02:15. We'll decide which request to accept or reject using the mathematical formulation:

$$Rate = R_p \times \frac{time\ frame - overlap\ time}{time\ frame} + R_c$$

Here, R_p is the number of requests in the previous window, which is 88. R_c is the number of requests in the current window, which is 12. The $time\ frame$ is 60 seconds in our case, and $overlap\ time$ is 15 seconds.

$$Rate = 88 \times \frac{60-15}{60} + 12$$

$$Rate = 78 < 100$$

As 78 is less than 100, so the incoming request is allowed.

Essential parameters

This algorithm is relatively more complex than the other algorithms described above. It requires the following parameters:

- **Rate limit** (R) It determines the number of maximum requests allowed per window.
- **Size of the window** (W)**:** This parameter represents the size of a time window that can be a minute, an hour, or any time slice.
- **The number of requests in the previous window** (R_p)**:** It determines the total number of requests that have been received in the previous time window.
- **The number of requests in the current window** (R_c)**:** It represents the number of requests received in the current window.
- **Overlap time** (O_t)**:** This parameter shows the overlapping time of the rolling window with the current window.

Advantages

- The algorithm is also space efficient due to limited states: the number of requests in the current window, the number of requests in the previous window, the overlapping percentage, and so on.
- It smooths out the bursts of requests and processes them with an approximate average rate based on the previous window.

Disadvantages

- This algorithm assumes that the number of requests in the previous window is evenly distributed, which may not always be possible.

A comparison of rate-limiting algorithms

The two main factors that are common among all the rate-limiting algorithms are:

- **Memory:** This feature refers to the number of states an algorithm requires to maintain for a normal operation. For example, if one

algorithm requires fewer variables (states) than the other, it is more space efficient.
- **Burst:** This refers to an increase of traffic in a unit time exceeding the defined limit.

The following table shows the space efficiency and burst of traffic for all algorithms that have been described in this lesson.

A Comparison of Rate-limiting Algorithms

Algorithm	Space efficient	Allows burst?
Token bucket	Yes	Yes, it allows a burst of traffic within defined limit.
Leaking bucket	Yes	No
Fixed window counter	Yes	Yes, it allows bursts at the edge of the time window and can exceed the

Requirements of a Blob Store's Design

Identify the requirements of and make estimations for the blob store.

We'll cover the following

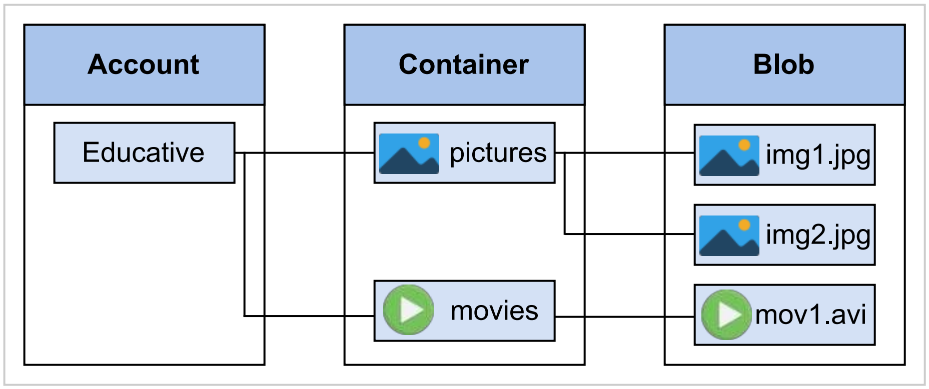
Requirements

Let's understand the functional and non-functional requirements below:

Functional requirements

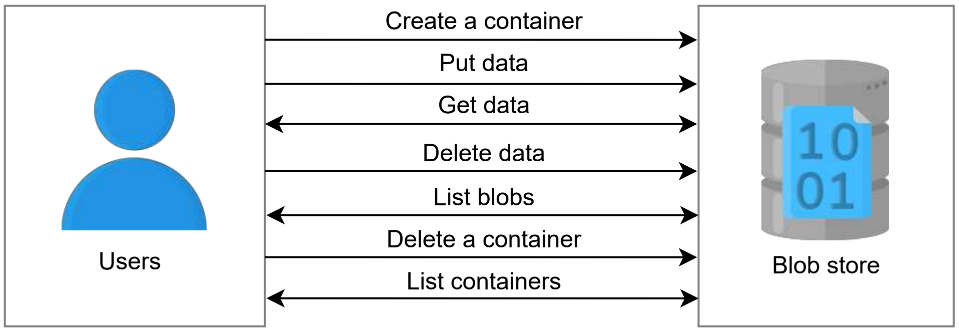
Here are the functional requirements of the design of a blob store:

- **Create a container**: The users should be able to create containers in order to group blobs. For example, if an application wants to store use specific data, it should be able to store blobs for different user account in different containers. Additionally, a user may want to group video blobs and separate them from a group of image blobs. A single blob store user can create many containers, and each container can have many blobs, as shown in the following illustration. For the sake of simplicity, we assume that we can’t create a container inside a container.



Multiple containers associated with a single storage account, and multiple blobs inside a single container

- **Put data:** The blob store should allow users to upload blobs to the created containers.
- **Get data:** The system should generate a URL for the uploaded blob, so that the user can access that blob later through this URL.
- **Delete data:** The users should be able to delete a blob. If the user wants to keep the data for a specified period of time (retention time), our system should support this functionality.
- **List blobs:** The user should be able to get a list of blobs inside a specific container.
- **Delete a container:** The users should be able to delete a container and all the blobs inside it.
- **List containers:** The system should allow the users to list all the containers under a specific account.

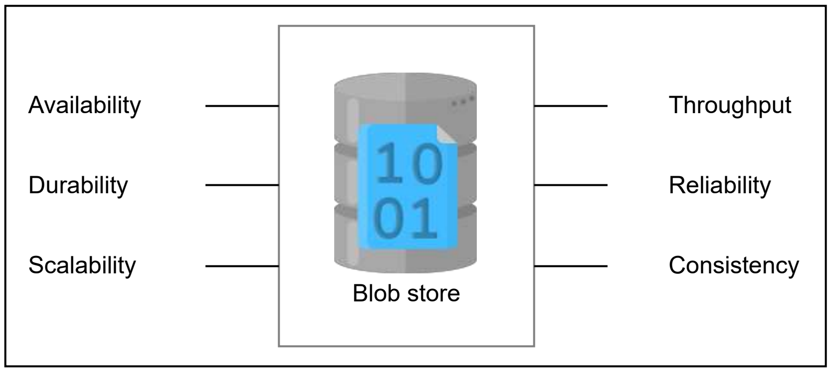


Functional requirements of a blob store

Non-functional requirements

Here are the non-functional requirements of a blob store system:

- **Availability:** Our system should be highly available.
- **Durability:** The data, once uploaded, shouldn't be lost unless users explicitly delete that data.
- **Scalability:** The system should be capable of handling billions of blobs.
- **Throughput**: For transferring gigabytes of data, we should ensure a high data throughput.
- **Reliability:** Since failures are a norm in distributed systems, our design should detect and recover from failures promptly.
- **Consistency:** The system should be strongly consistent. Different users should see the same view of a blob.



The non-functional requirements of a blob store

Resource estimation

Let’s estimate the total number of servers, storage, and bandwidth required by a blob storage system. Because blobs can have all sorts of data, mentioning all of those types of data in our estimation may not be practical. Therefore, we’ll use YouTube as an example, which stores videos and thumbnails on the blob store. Furthermore, we’ll make the following assumptions to complete our estimations.

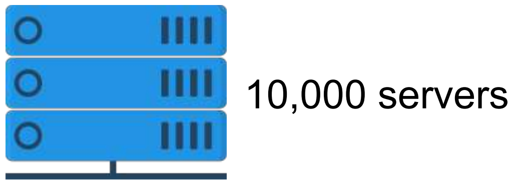
Assumptions:

- The number of daily active users who upload or watch videos is five million.
- The number of requests per second that a single blob store server can handle is 500.
- The average size of a video is 50 MB.
- The average size of a thumbnail is 20 KB.
- The number of videos uploaded per day is 250,000.
- The number of read requests by a single user per day is 20.

Number of servers estimation

From our assumptions, we use the number of daily active users (DAUs) and queries a blob store server can handle per second. The number of servers that we require is calculated using the formula given below:

$$\frac{Number\ of\ active\ users}{Queries\ handled\ per\ server} = 10K\ servers$$



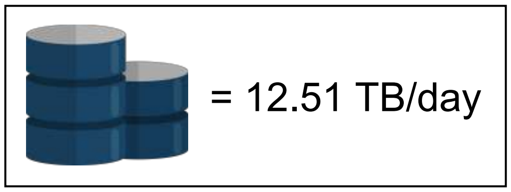
Number of servers required by a blob store system dedicated to storing YouTube data

Storage estimation

Considering the assumptions written above, we use the formula given below to compute the total storage required by YouTube in one day:

$$Total_{storage/day} = No.\ of\ videos_{/day} \times (Storage_{/video} + Storage_{/thumbnail})$$

Putting the numbers from above into the formula gives us $12.51\ TB_{/day}$, which is the approximate storage required by YouTube per day for keeping a single copy of the uploaded video in a single resolution.



Total Storage Required to Store Videos and Thumbnails Uploaded Per Day on YouTube

No. of videos per day	Storage per video (MB)	Storage per thumbnail (KB)	Total storage per day (TB)
250000	50	20	*f* 12.51

Bandwidth estimation

Let's estimate the bandwidth required for uploading data to and retrieving data from the blob store.

Incoming traffic: To estimate the bandwidth required for incoming traffic, we consider the total data uploaded per day, which indirectly means the total storage needed per day that we calculated above. The amount of data transferred to the servers per second can be computed using the following formula:

$$Total_{bandwidth} = \frac{Total_{storage\_day}}{24 \times 60 \times 60}$$

Bandwidth Required for Uploading Videos on YouTube

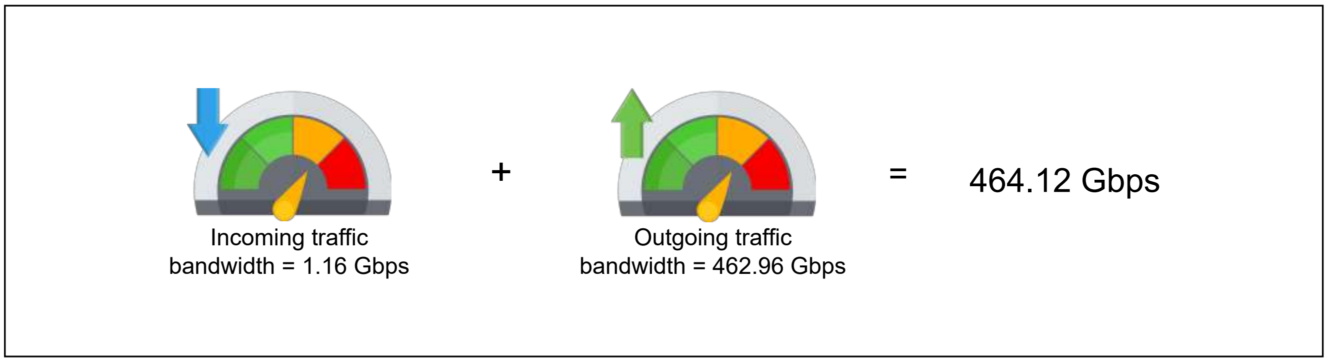
Total storage per day (TB)	Seconds in a day	Bandwidth (Gb/s)
12.51	86400	*f* 1.16

Outgoing traffic: Since the blob store is a read-intensive store, most of the bandwidth is required for outgoing traffic. Considering the aforementioned assumptions, we calculate the bandwidth required for outgoing traffic using the following formula:

$$Total_{bandwidth} = \frac{No.\ of\ active\ users_{/day} \times No.\ of\ requests_{/user/day} \times Total_{data\_s}}{Seconds\ in\ a\ day}$$

Bandwidth Required for Downloading Videos on YouTube

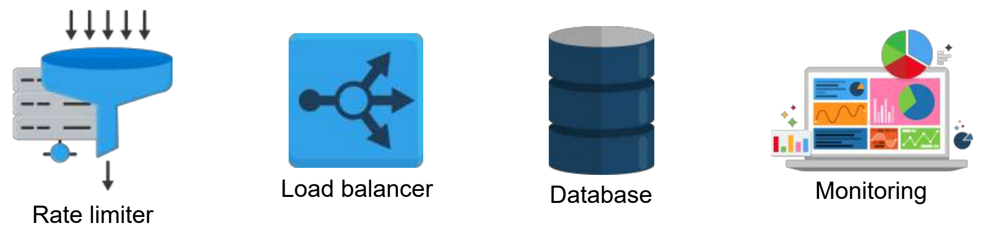
No. of active users per day	No. of requests per user	Data size (MB)	Bandwidth required (Gb/s)
5000000	20	50	*f* 462.96


Incoming traffic bandwidth = 1.16 Gbps
+
Outgoing traffic bandwidth = 462.96 Gbps
=
464.12 Gbps

Summarizing the bandwidth requirements of a blob store system for YouTube videos only

Building blocks we will use

We use the following building blocks in the design of our blob store system:



Building blocks for the design of a task scheduler

- **Rate Limiter:** A rate limiter is required to control the users' interaction with the system.
- **Load balancer:** A load balancer is needed to distribute the request load onto different servers.
- **Database:** A database is used to store metadata information for the blobs.
- **Monitoring:** Monitoring is needed to inspect storage devices and the space available on them in order to add storage on time if needed.

In this lesson, we discussed the requirements and estimations of the blob store system. We'll design the blob store system in the next lesson, all while following the delineated requirements.

← Back

Next →

System Design: A Blob Store

Design of a Blob Store

✓ Mark as Completed

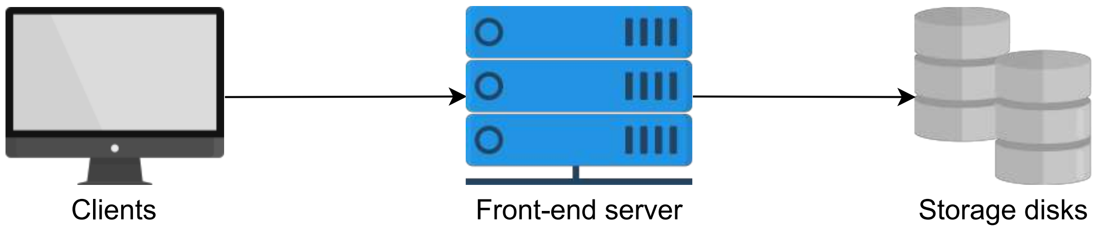
Design of a Blob Store

Learn how to incorporate certain requirements into the design of a blob store.

We'll cover the following

High-level design

Let's identify and connect the components of a blob store system. At a high level, the components are clients, front-end servers, and storage disks.



The high-level design of a blob store

The client's requests are received at the front-end servers that process the request. The front-end servers store the client's blob onto the storage disks attached to them.

API design

Let’s look into the API design of the blob store. All of the functions below can only be performed by a registered and authenticated user. For the sake of brevity, we don’t include functionalities like registration and authentication of users.

Create container

The `createContainer` operation creates a new container under the logged-in account from which this request is being generated.

```
createContainer(containerName)
```

Parameter	Description
`containerName`	This is the name of the container. It should be unique within a storage

Upload blobs

The client’s data is stored in the form of Bytes in the blob store. The data can be put into a container with the following code:

```
putBlob(containerPath, blobName, data)
```

Parameter	Description
`containerPath`	This is the path of the container in which we upload the blob. It cor and `containerID` .
`blobName`	This is the name of the blob. It should be unique within a containe will give the blob that was uploaded later a version number.
`data`	This is a file that the user wants to upload to the blob store.

> **Note:** This API is just a logical way to spell out needs. We might use a multistep streaming call for actual implementation if the data size is very large.

Download blobs

Blobs are identified by their unique name or ID.

```
getBlob(blobPath)
```

Parameter	Description
`blobPath`	This is the fully qualified path of the data or file, including its unique I

Delete blob

The `deleteBlob` operation marks the specified blob for deletion. The actual blob is deleted during garbage collection.

```
deleteBlob(blobPath)
```

Parameter	Description
`blobPath`	This is the path of the blob that the user wants to delete.

List blobs

The `listBlobs` operation returns a list of blobs under the specified container or path.

```
listBlobs(containerPath)
```

Parameter	Description
`containerPath`	This is the path to the container from which the user wants to get the l

Delete container

The `deleteContainer` operation marks the specified container for deletion. The container and any blobs in it are deleted later during garbage collection.

```
deleteContainer(containerPath)
```

Parameter	Description
`containerPath`	This is the path to the container that the user wants to delete.

List containers

The `listContainers` operation returns a list of the containers under the specified user's blob store account.

```
listContainers(accountID)
```

Parameter	Description

`accountID`	This is the ID of the user who wants to list their containers.

> Note: The APIs used to retrieve blobs provide metadata containing size, version number, access privileges, name, and so on.

Detailed design

We start this section by identifying the key components that we need to complete our blob store design. Then, we look at how these components connect to fulfill our functional requirements.

Components

Here is a list of components that we use in the blob store design:

- **Client:** This is a user or program that performs any of the API functions that are specified.
- **Rate limiter:** A rate limiter limits the number of requests based on the user's subscription or limits the number of requests from the same IP address at the same time. It doesn't allow users to exceed the predefined limit.
- **Load balancer:** A load balancer distributes incoming network traffic among a group of servers. It's also used to reroute requests to different regions depending on the location of the user, different data centers within the same region, or different servers within the same data center. **DNS** load balancing can be used to reroute the requests amon different regions based on the location of the user.
- **Front-end servers:** Front-end servers forward the users' requests for adding or deleting data to the appropriate storage servers.

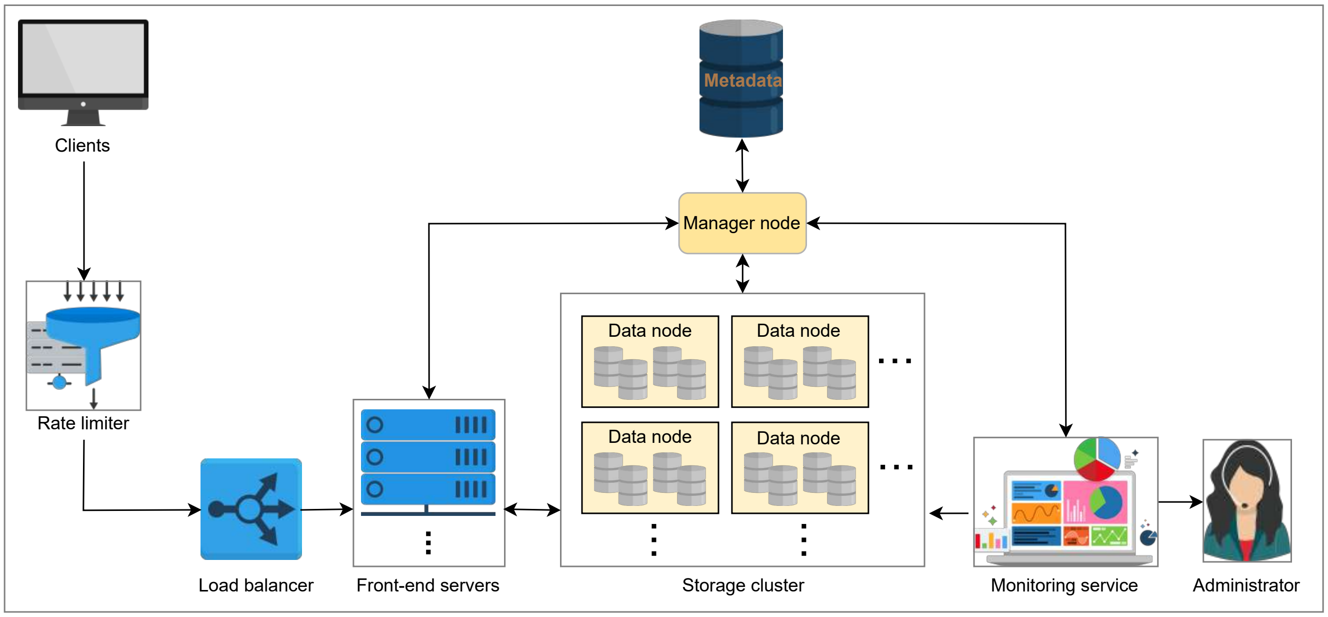
- **Data nodes:** Data nodes hold the actual blob data. It's also possible that they contain a part of the blob's data. Blobs are split into small, fixed-size pieces called **chunks**. A data node can accommodate all of the chunks of a blob or at least some of them.
- **Manager node:** A manager node is the core component that manages all data nodes. It stores information about storage paths and the access privileges of blobs. There are two types of access privileges: private and public. A *private* access privilege means that the blob is only accessible by the account containing that blob. A *public* access privilege means that anyone can access that blob.

> **Note:** Each of the data nodes in the cluster send the manager node a heartbeat and a chunk report regularly. The presence of a **heartbeat** indicates that the data node is operational. A **chunk report** lists all the chunks on a data node. If a data node fails to send a heartbeat, the manager node considers that node dead and then processes the user requests on the replica nodes. The manager node maintains a log of pending operations that should be replayed on the dead data node when it recovers.

- **Metadata storage:** Metadata storage is a distributed database that's used by the manager node to store all the metadata. Metadata consists of account metadata, container metadata, and blob metadata.
    - ***Account metadata*** contains the account information for each user and the containers held by each account.
    - ***Container metadata*** consists of the list of the blobs in each container.
    - ***Blob metadata*** consists of where each blob is stored. The blob metadata is discussed in detail in the next lesson.

- **Monitoring service:** A monitoring service monitors the data nodes and the manager node. It alerts the administrator in case of disk failures that require human intervention. It also gets information about the total available space left on the disks to alert administrators to add more disks.
- **Administrator:** An administrator is responsible for handling notifications from the monitoring services and conducting routine checkups of the overall service to ensure reliability.

The architecture of how these components interconnect is shown in the diagram below:



The detailed design of a blob store

Workflow

We describe the workflow based on the basic operations we can perform on a blob store. We assume that the user has successfully logged in and a container has already been created. A unique ID is assigned to each user and container. The user performs the following operations in a specific container.

Write a blob

1. The client generates the upload blob request. If the client's request successfully passes through the rate limiter, the load balancer forwards the client's request to one of the front-end servers. The front-end server then requests the manager node for the data nodes it should contact to store the blob.
2. The manager node assigns the blob a unique ID using a unique ID generator system. It then splits the large-size blob into smaller, fixed-size chunks and assigns each chunk a data node where that chunk is eventually stored. The manager node determines the amount of storage space that's available on the data nodes using a **free-space management system**.
3. After determining the mapping of chunks to data nodes, the front-end servers write the chunks to the assigned data nodes.
4. We replicate each chunk for redundancy purposes. All choices regarding chunk replication are made at the manager node. Hence, the manager node also allocates the storage and data nodes for storing replicas.
5. The manager node stores the blob metadata in the metadata storage. We discuss the blob's metadata schema in detail in the next lesson.
6. After writing the blob, a fully qualified path of the blob is returned to the client. The path consists of the user ID, container ID where the user has added the blob, the blob ID, and the access level of the blob.

Point to Ponder

Question

What does the manager node do if a user concurrently writes two blobs with the same name inside the same container?

Show Answer

Reading a blob

1. When a read request for a blob reaches the front-end server, it asks the manager node for that blob's metadata.
2. The manager node first checks whether that blob is private or public, based on the path of the blob and whether we're authorized to transfer that blob or not.
3. After authorizing the blob, the manager node looks for the chunks for that blob in the metadata and looks at their mappings to data nodes. The manager node returns the chunks and their mappings (data nodes) to the client.
4. The client then reads the chunk data from the data nodes.

> **Note:** The metadata information for reading the blob is cached at the client machine, so that the next time a client wants to read the same blob, we won't have to burden the manager node. Additionally, the client's read operation will be faster the next time.

Point to Ponder

Question

Suppose the manager node moves data from one data node to another because of an impending disk failure. The user will now have stale information if they use the cached metadata to access th data. How do we handle such situations?

Show Answer

Deleting a blob Upon receiving a delete blob request, the manager node marks that blob as deleted in the metadata, and frees up the space later using a garbage collector. We learn more about garbage collectors in the next lesson.

Point to Ponder

Question

Can the manager node be considered a single point of failure? If yes, then how can we cope with this problem?

Show Answer

In the next lesson, we talk about the design considerations of a blob store.

Back
Requirements of a Blob Store's Design

Next
Design Considerations of a Blob S

✓ Mark as Comple

Design Considerations of a Blob Store

Learn more details about the different design aspects of the blob store.

We'll cover the following

Introduction

Even though we discussed the design of the blob store system and its major components in detail in the previous lesson, a number of interesting questions still require answers. For example, how do we store large blobs? Do we store them in the same disk, in the same machine, or do we divide those blobs into chunks? How many replicas of a blob should be made to ensure reliability and availability? How do we search for and retrieve blobs quickly?. These are just some of the questions that might come up.

This lesson addresses these important design concerns. The table below summarizes the goals of this lesson.

Summary of the Lesson

Section	Purpose
Blob metadata	This is the metadata that's maintained to ensure efficient storage a retrieval of blobs.
Partitioning	This determines how blobs are partitioned among different data n
Blob indexing	This shows us how to efficiently search for blobs.
Pagination	This teaches us how to conceive a method for the retrieval of a lim number of blobs to ensure improved readability and loading time.
Replication	This teaches us how to replicate blobs and tells us how many copie should maintain to improve availability.
Garbage collection	This teaches us how to delete blobs without sacrificing performanc
Streaming	This teaches us how to stream large files chunk-by-chunk to facilita interactivity for users.
Caching	This shows us how to improve response time and throughput.

Before we answer the questions listed above, let's look at how we create layers of abstractions for the user to hide the internal complexity of a blob store. These abstraction layers help us make design-related decisions as well.

There are three layers of abstractions:

1. **User account**: Users uniquely get identified on this layer through thei `account_ID`. Blobs uploaded by users are maintained in their containers

2. **Container**: Each user has a set of containers that are all uniquely identified by a `container_ID`. These containers contain blobs.
3. **Blob**: This layer contains information about blobs that are uniquely identified by their `blob_ID`. This layer maintains information about the metadata of blobs that's vital for achieving the availability and reliability of the system.

We can take routing, storage, and sharding decisions on the basis of these layers. The table below summarizes these layers.

Layered Information

Level	Uniquely identified by	Information	Sharded
User's blob store account	`account_ID`	list of `containers_ID` values	`account_`
Container	`container_ID`	List of `blob_ID` values	`container`
Blob	`blob_ID`	{list of chunks, chunkInfo: data node ID's,.. }	`blob_I`

> **Note:** We generate unique IDs for user accounts, containers, and blobs using a unique ID generator.

Besides storing the actual blob data, we have to maintain some metadata for managing the blob storage. Let's see what that data is.

Blob metadata

When a user uploads a blob, it's split into small-sized chunks in order to be able to support the storage of large files that can't fit in one contiguous location, in one data node, or in one block of a disk associated with that data node. The chunks for a single blob are then stored on different data nodes that have enough storage space available to store these chunks. There are billions of blobs that are kept in storage. The manager node has to store all the information about the blob's chunks and where they are stored, so that it can retrieve the chunks on reads. The manager node assigns an ID to each chunk.

The information about a blob consists of chunk IDs and the name of the assigned data node for each chunk. We split the blobs into equal-sized chunks. Chunks are replicated to enable them to deal with data node failure. Hence, we also store the replica IDs for each chunk. We have access to all this information pertaining to each blob.

Let's say we have a blob of 128 MB, and we split it into two chunks of 64 MB each. The metadata for this blob is shown in the following table:

Blob Metadata

Chunk	Datanode ID	Replica 1 ID	Replica 2 ID	Replica 3 ID
1	d1b1	r1b1	r2b1	r3b1
2	d1b2	r1b2	r2b2	r3b2

> **Note:** To avoid complexity, the chunk size is fixed for all the blobs in a blob store. The chunk size depends on the performance requirements of a blob store. We desire a larger chunk size to

> maintain small metadata at the manager node because a large chunk size results in a higher disk latency, which leads to slower performance. Interestingly, disks can have almost the same latency for reading and writing a range of data. For example, a disk can have similar latency for writing MBs in the range of 4–8. Additionally, they can have similar latency for writing data in the range of 9–20 MBs. This is due to the use of contiguous sectors on the disks, and caching on the disk and on the server by its OS.

We maintain three replicas for each block. When writing a blob, the manager node identifies the data and the replica nodes using its free space management system. Besides handling data node failure, the replica nodes are also used to serve read/write requests, so that the primary node is not overloaded.

In the example above, the blob size is a multiple of the chunk size, so the manager node can determine how many Bytes to read for each chunk.

Point to Ponder

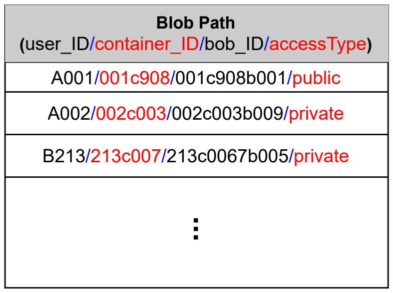
Question

What if the blob size isn't a multiple of our configured chunk size? How does the manager node know how many Bytes to read for the last chunk?

Show Answer

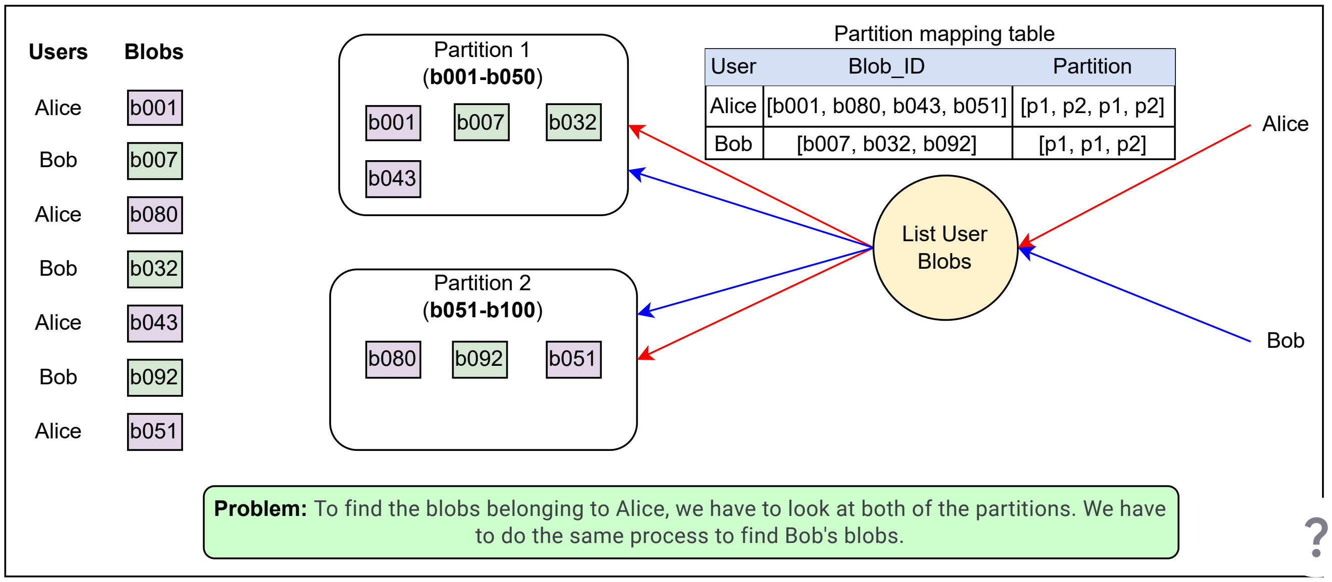
Partition data

We talked about the different levels of abstraction in a blob store—the account layer, the container layer, and the blob layer. There are billions of blobs that are stored and read. There is a large number of data nodes on which we store these blobs. If we look for the data nodes that contain specific blobs out of all of the data nodes, it would be a very slow process. Instead, we can group data nodes and call each group a **partition**. We maintain a partition map table that contains a list of all the blobs in each partition. If we distribute the blobs on different partitions independent of their container IDs and account IDs, we encounter a problem, as shown in the following illustration:



Blob Path (user_ID/container_ID/bob_ID/accessType)
A001/001c908/001c908b001/public
A002/002c003/002c003b009/private
B213/213c007/213c0067b005/private
⋮

Billions of blobs



Users	Blobs
Alice	b001
Bob	b007
Alice	b080
Bob	b032
Alice	b043
Bob	b092
Alice	b051

Partition mapping table

User	Blob_ID	Partition
Alice	[b001, b080, b043, b051]	[p1, p2, p1, p2]
Bob	[b007, b032, b092]	[p1, p1, p2]

Problem: To find the blobs belonging to Alice, we have to look at both of the partitions. We have to do the same process to find Bob's blobs.

Range partitioning based on blob IDs

Partitioning based on the blob IDs causes certain problems. For example, tʰ
blobs under a specific container or account may reside in different

partitions that add overhead while reading or listing the blobs linked to a particular account or a particular container.

To overcome the problem described above, we can partition the blobs based on the complete path of the blob. The partition key here is the combination of the account ID, container ID, and blob ID. This helps in co-locating the blobs for a single user on the same partition server, which enhances performance.

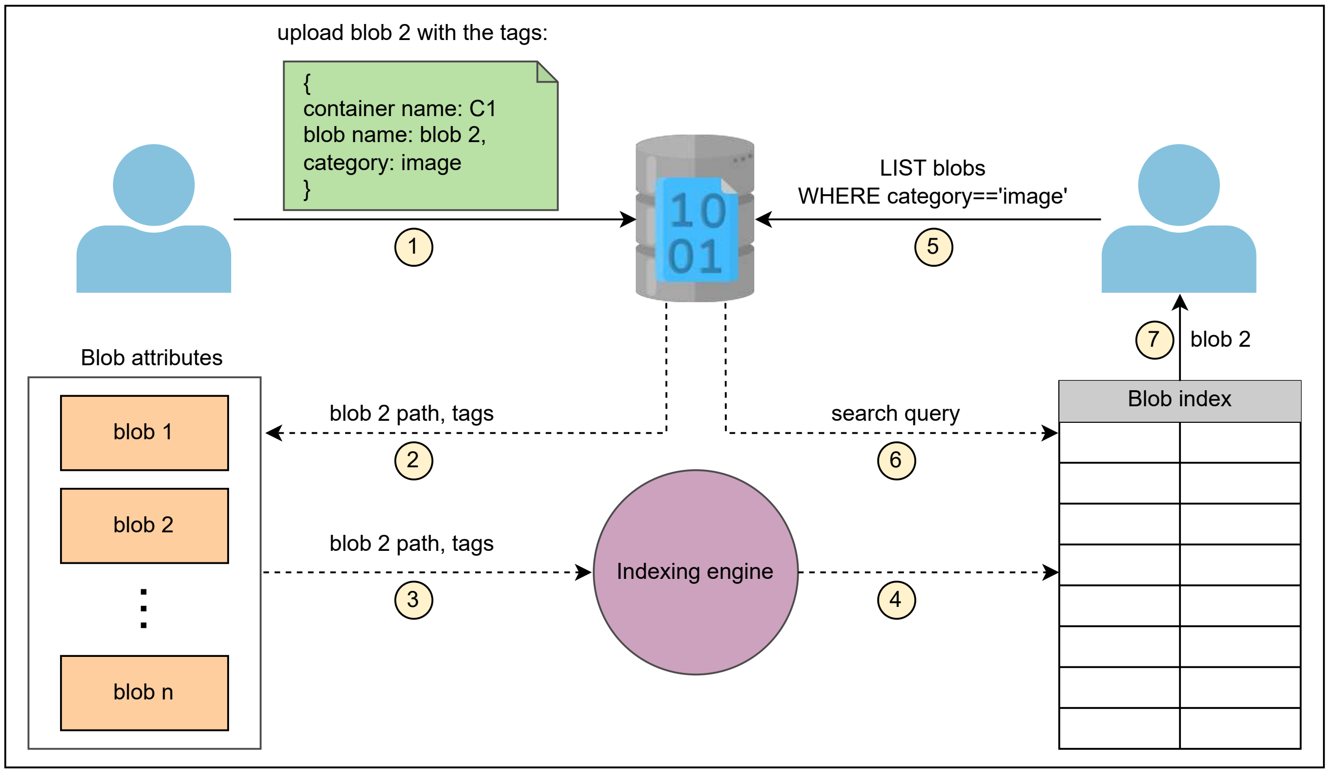
> **Note:** The **partition mappings** are maintained by the manager node, and these mappings are stored in the distributed metadata storage.

Blob indexing

Finding specific blobs in a sea of blobs becomes more difficult and time-consuming with an increase in the number of blobs that are uploaded to the storage. The **blob index** solves the problem of blob management and querying.

To populate the blob index, we define key-value tag attributes on the blobs while uploading the blobs. We use multiple tags, like container name, blob name, upload date and time, and some other categories like the image or video blob, and so on.

As shown in the following illustration, a blob indexing engine reads the new tags, indexes them, and exposes them to a searchable blob index:



Indexing and searching blobs

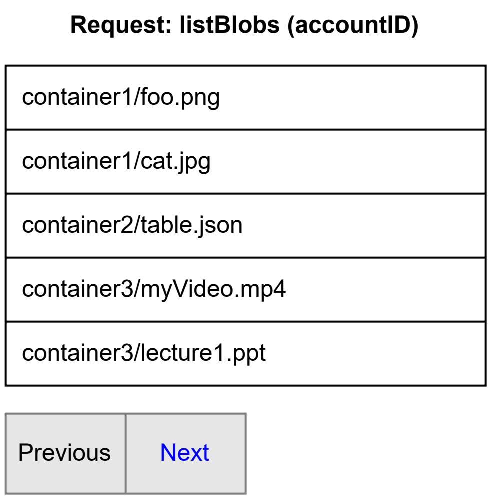
We can categorize blobs as well as sort blobs using indexing. Let's see how we utilize indexing in pagination.

Pagination for listing

Listing is about returning a list of blobs to the user, depending on the user's entered prefix. A **prefix** is a character or string that returns the blobs whose name begins with that particular character or string.

Users may want to list all the blobs associated with a specific account, all the blobs present inside a specific container, or they may want to list some public blobs based on a prefix. The problem is that this list could be very long. We can't return the whole list to the user in one go. So, we have to return the list of the blobs in parts.

Let's say a user wants a list of blobs associated with their account and there are a total of 2,000 blobs associated with that account. Searching, returning, and loading too many blobs at once affects performance. This is where paging becomes important. We can return the first five results and give users a `next` button. On each click of the `next` button, it returns the next five



The application owners set the number of results to return depending on these factors:

- How much time they assume the users should wait for query response.
- How many results they can return in that time. We have shown five results per page, which is a very small number. We use this number just for visualization purposes.

Point to Ponder

Question

How do we decide which five blobs to return first out of the 2,000 blobs totalt?

Show Answer

For pagination, we need a **continuation token** as a starting point for the part of the list that's returned next. A continuation token is a string token that's included in the response of a query if the total number of queried results exceeds the maximum number of results that we can return at once. As a result, it serves as a pointer, allowing the re-query to pick up where we left off.

Replication

Replication is carried out on two levels to support availability and strong consistency. To keep the data strongly consistent, we synchronously replicate data among the nodes that are used to serve the read requests, right after performing the write operation. To achieve availability, we can replicate data to different regions or data centers after performing the write operation. We don't serve the read request from the other data center or regions until we have not replicated data there.

These are the two levels of replication:

- *Synchronous replication* within a storage cluster.
- *Asynchronous replication* across data centers and regions.

Synchronous replication within a storage cluster

A **storage cluster** is made up of N racks of storage nodes, each of which is configured as a fault domain with redundant networking and power.

We ensure that every data written into a storage cluster is kept durable within that storage cluster. The manager node maintains enough data replicas across the nodes in distinct fault domains to ensure data durabilit inside the cluster in the event of a disk, node, or rack failure.

> **Note:** This intra-cluster replication is done on the critical path of the client's write requests.

Success can be returned to the client once a write has been synchronously replicated inside that storage cluster. This allows for **quick writes** because:

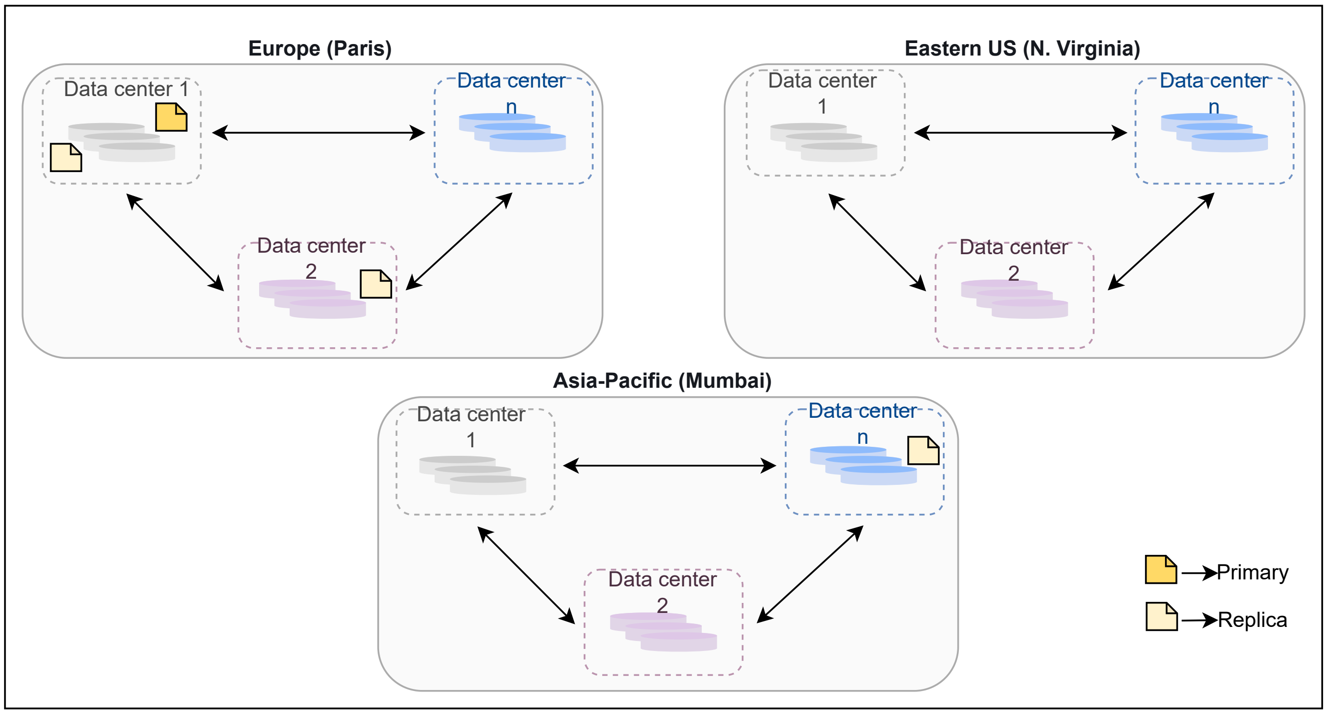
- We replicate data within a storage cluster where all the nodes are nearby, thus reducing the latency.
- We use in-lined data copying where we use redundant network paths to copy data to all the replicas in parallel.

This replication technique helps maintain data consistency and availability inside the storage cluster.

Asynchronous replication across data centers and region

The blob store's data centers are present in different regions—for example, Asia-Pacific, Europe, eastern US, and more. In each region, we have more than one data center placed at different locations, so that if one data center goes down, we have other data centers in the same region to step in and serve the user requests. There are a minimum of three data centers within each region, each separated by miles to protect against local events like fires, floods, and so on.

The number of copies of a blob is called the **replication factor**. Most of the time, a replication factor of **three** is sufficient.



These are the regions and availability zones. Dark yellow is the primary data and light yellow are the replicas

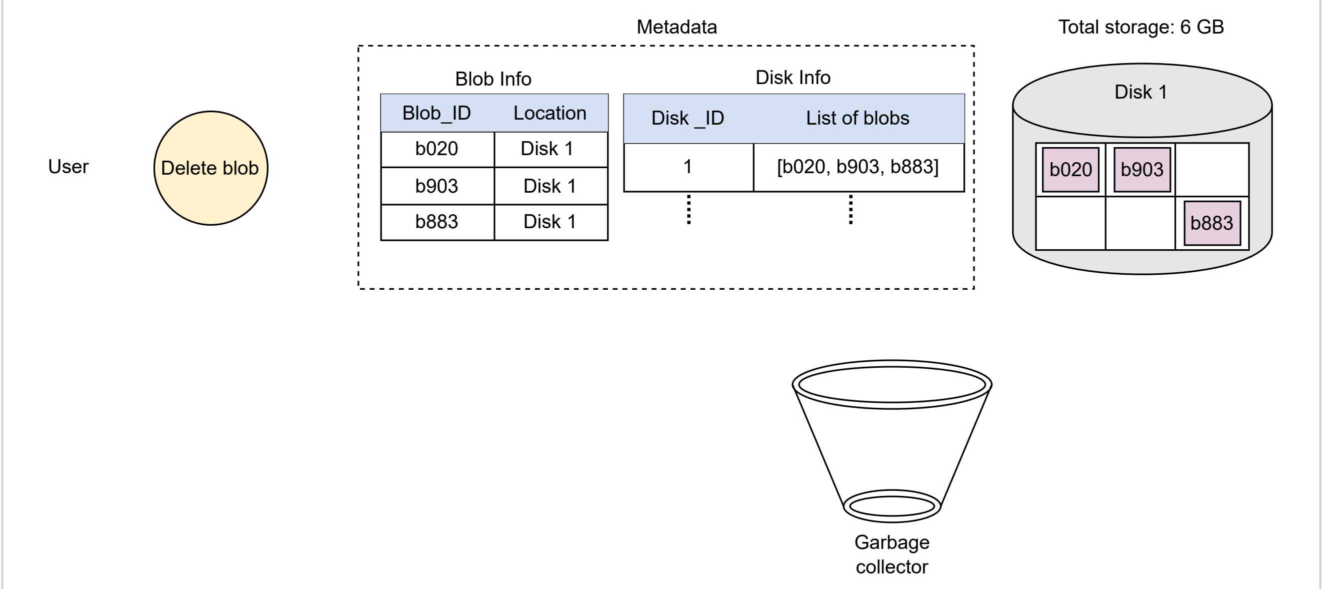
We keep four copies of a blob. One is the local copy within the data center in the primary region to protect against server rack and drive failures. The second copy of the blob is placed in the other data center within the same region to protect against fire or flooding in the data center. The third copy is placed in the data center of a different region to protect against regional disasters.

Garbage collection while deleting a blob

Since the blob chunks are placed at different data nodes, deleting from many different nodes takes time, and holding a client until that's done is not a viable option. Due to real-time latency optimization, we don't actually remove the blob from the blob store against a delete request. Instead, we just mark a blob as "DELETED" in the metadata to make it inaccessible to the user. The blob is removed later after responding to the user's delete reque

Marking the blob as deleted, but not actually deleting it at the moment, causes internal metadata inconsistencies, meaning that things continue to take up storage space that should be free. These metadata inconsistencies have no impact on the user. For example, for a blob marked as deleted in the metadata, we still have the entries for that blob's chunks in the metadata. The data nodes are still holding on to that blob's chunks. Therefore, we have a service called a **garbage collector** that cleans up metadata inconsistencies later. The deletion of a blob causes the chunks associated with that blob to be freed. However, there could be an appreciable time delay between the time a blob is deleted by a user and the time of the corresponding increase in free space in the blob store. We can bear this appreciable time delay because, in return, we have a real-time fast response benefit for the user's delete blob request.

The whole deletion process is shown in the following illustration:



Blob Info

Blob_ID	Location
b020	Disk 1
b903	Disk 1
b883	Disk 1

Disk Info

Disk _ID	List of blobs
1	[b020, b903, b883]
⋮	⋮

Three blobs are uploaded to the blob store. Disk 1 is holding all three blobs

1 of 10

Stream a file

To stream a file, we need to define how many Bytes are allowed to be read at one time. Let's say we read X number of Bytes each time. The first time we read the first X Bytes starting from the 0th Byte (0 to $X - 1$) and the next time, we read the next X Bytes (X to $2X - 1$).

Point to Ponder

Question

How do we know which Bytes we have read first and which Bytes we have to read next?

Show Answer

Cache the blob store

Caching can take place at multiple levels. Below are some examples:

- The metadata for a blob's chunks is cached on the client side when it's read for the first time. The client can go directly to the data nodes without communicating to the manager node to read the same blob a second time.
- At the **front-end servers**, we cache the partition map and use it to determine which partition server to forward each request to.
- The frequently accessed chunks are cached at the **manager node**, which helps us stream large objects efficiently. It also reduces disk I/O.