What kind of code structure could allow this disastrous bug to slip through? We could only guess. We never received a full explanation.

What are some of the most impactful software bugs you encountered in your career?

AWS Lambda behind the scenes

Serverless is one of the hottest topics in cloud services. How does AWS Lambda work behind the scenes?

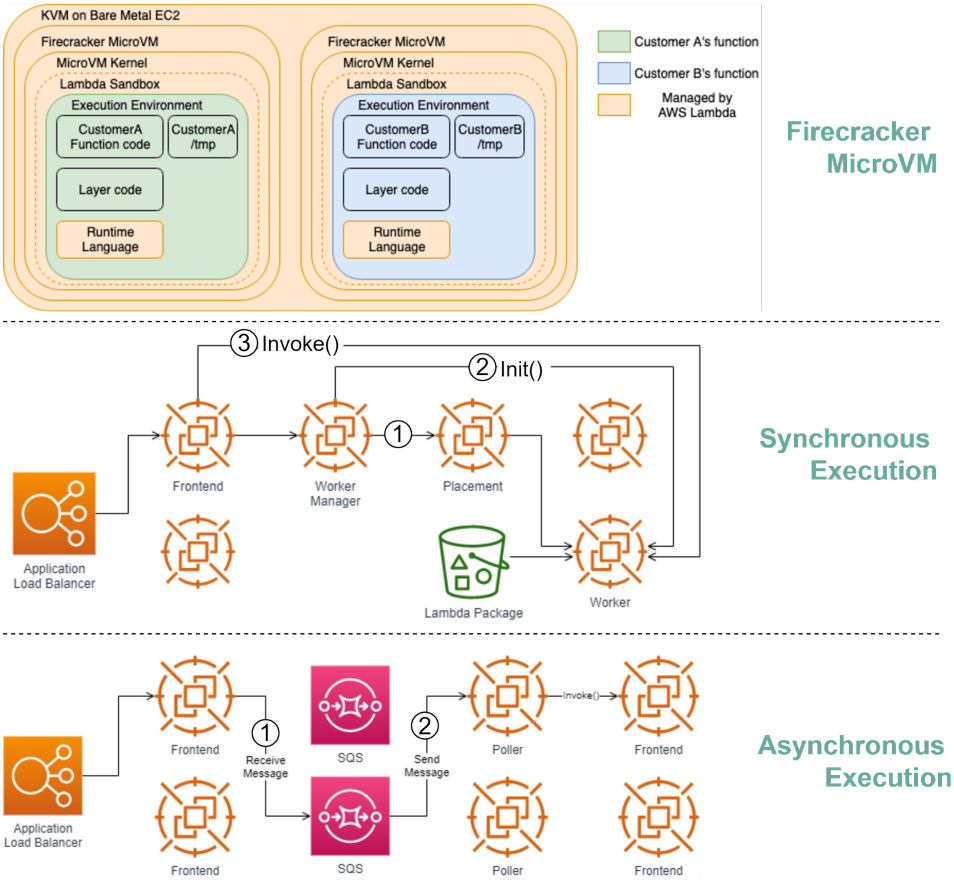
Lambda is a **serverless** computing service provided by Amazon Web Services (AWS), which runs functions in response to events.

Firecracker MicroVM

Firecracker is the engine powering all of the Lambda functions [1]. It is a virtualization technology developed at Amazon and written in Rust.

The diagram below illustrates the isolation model for AWS Lambda Workers.

How does AWS Lambda work?



Lambda functions run within a sandbox, which provides a minimal Linux userland, some common libraries and utilities. It creates the Execution environment (worker) on EC2 instances.

How are lambdas initiated and invoked? There are two ways.

Synchronous execution
Step1: "The Worker Manager communicates with a Placement Service which is responsible to place a workload on a location for the given host (it's provisioning the sandbox) and returns that to the Worker Manager" [2].

Step 2: "The Worker Manager can then call *Init* to initialize the function for execution by downloading the Lambda package from S3 and setting up the Lambda runtime" [2]

Step 3: The Frontend Worker is now able to call *Invoke* [2].

Asynchronous execution
Step 1: The Application Load Balancer forwards the invocation to an available Frontend which places the event onto an internal queue(SQS).
Step 2: There is "a set of pollers assigned to this internal queue which are responsible for polling it and moving the event onto a Frontend synchronously. After it's been placed onto the Frontend it follows the synchronous invocation call pattern which we covered earlier" [2].

Question: Can you think of any use cases for AWS Lambda?

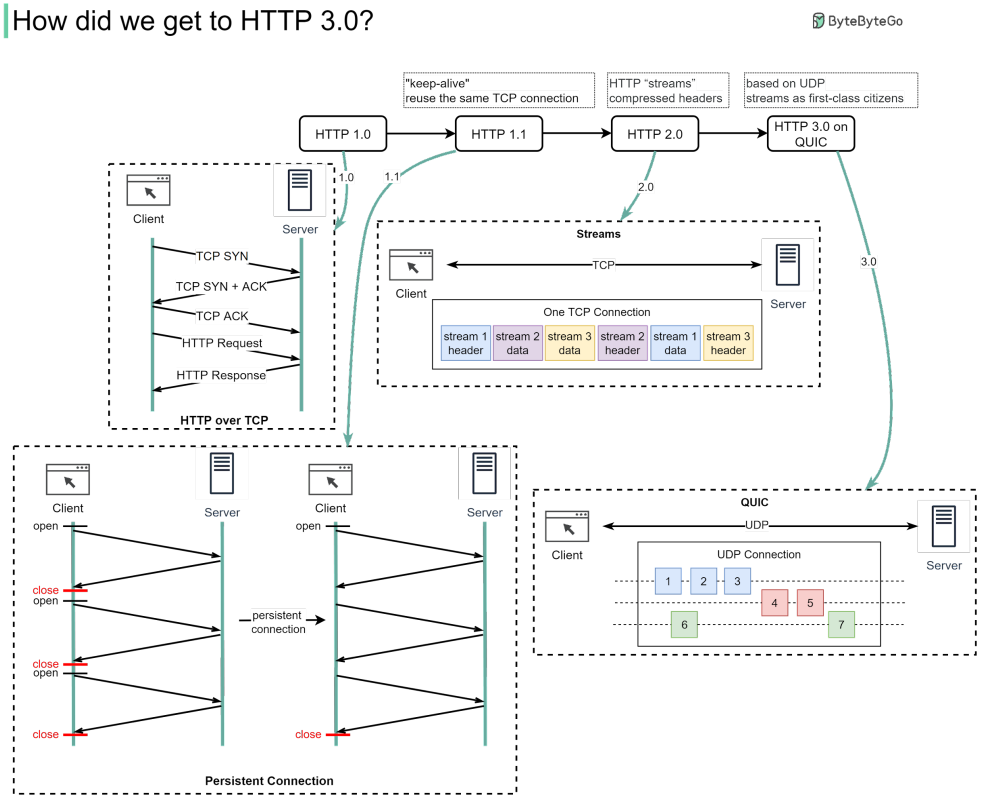
Sources:
[1] AWS Lambda whitepaper:
https://docs.aws.amazon.com/whitepapers/latest/security-overview-aws-lambda/lambda-executions.html
[2] Behind the scenes, Lambda:
https://www.bschaatsbergen.com/behind-the-scenes-lambda/

Image source: [1] [2]

HTTP 1.0 -> HTTP 1.1 -> HTTP 2.0 -> HTTP 3.0 (QUIC).

What problem does each generation of HTTP solve?

The diagram below illustrates the key features.



🔹 HTTP 1.0 was finalized and fully documented in 1996. Every request to the same server requires a separate TCP connection.

🔹 HTTP 1.1 was published in 1997. A TCP connection can be left open for reuse (persistent connection), but it doesn't solve the HOL (head-of-line) blocking issue.

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

🔹 HTTP 2.0 was published in 2015. It addresses HOL issue through request multiplexing, which eliminates HOL blocking at the application layer, but HOL still exists at the transport (TCP) layer.

As you can see in the diagram, HTTP 2.0 introduced the concept of HTTP “streams”: an abstraction that allows multiplexing different HTTP exchanges onto the same TCP connection. Each stream doesn’t need to be sent in order.

🔹 HTTP 3.0 first draft was published in 2020. It is the proposed successor to HTTP 2.0. It uses QUIC instead of TCP for the underlying transport protocol, thus removing HOL blocking in the transport layer.

QUIC is based on UDP. It introduces streams as first-class citizens at the transport layer. QUIC streams share the same QUIC connection, so no additional handshakes and slow starts are required to create new ones, but QUIC streams are delivered independently such that in most cases packet loss affecting one stream doesn't affect others.

Question: When shall we upgrade to HTTP 3.0? Any pros & cons you can think of?

—

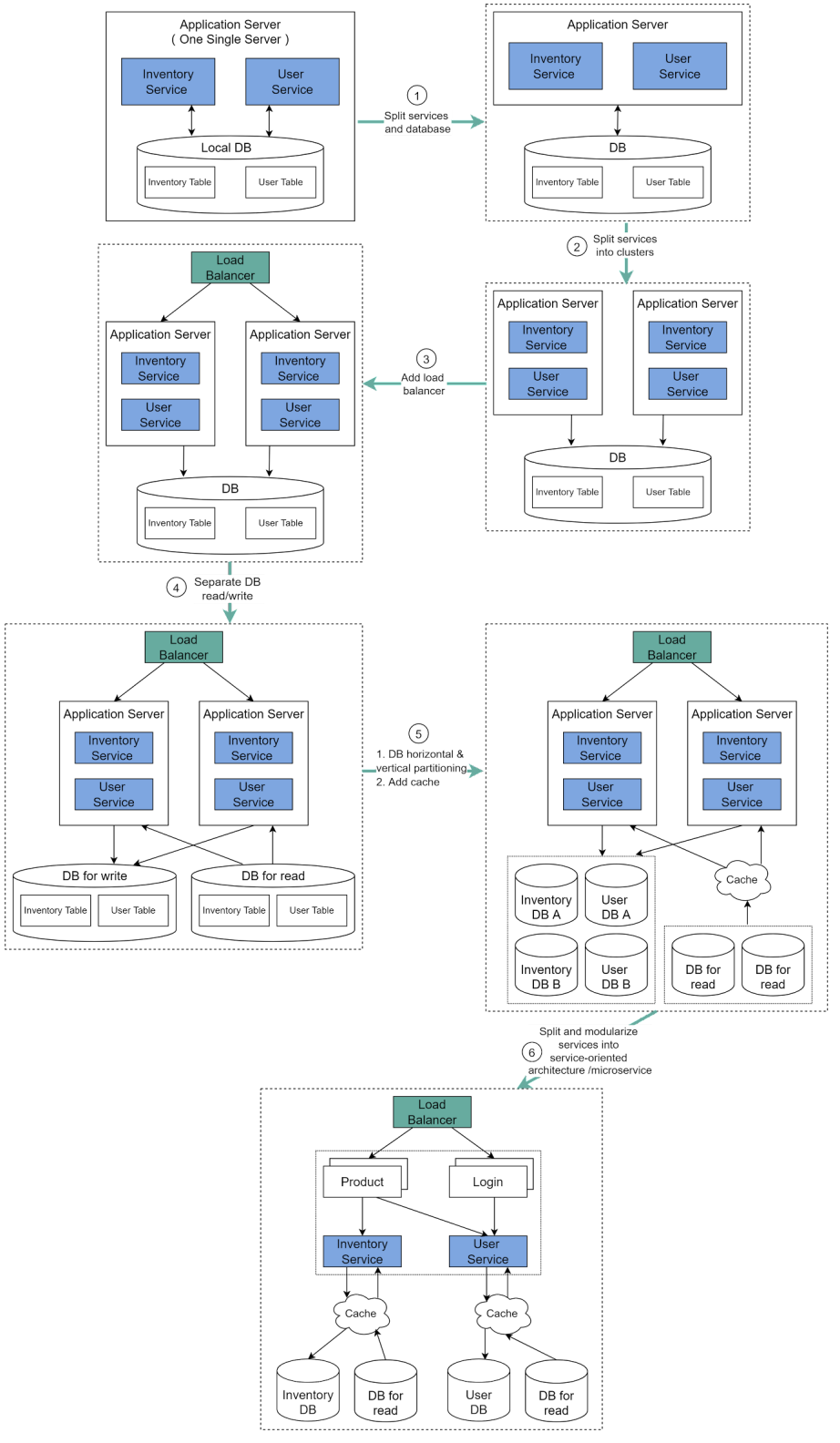
How to scale a website to support millions of users?

We will explain this step-by-step.

The diagram below illustrates the evolution of a simplified eCommerce website. It goes from a monolithic design on one single server, to a service-oriented/microservice architecture.

How to Scale a Website Step-by-Step?

ByteByteGo



Suppose we have two services: inventory service (handles product descriptions and inventory management) and user service (handles user information, registration, login, etc.).

Step 1 - With the growth of the user base, one single application server cannot handle the traffic anymore. We put the application server and the database server into two separate servers.

Step 2 - The business continues to grow, and a single application server is no longer enough. So we deploy a cluster of application servers.

Step 3 - Now the incoming requests have to be routed to multiple application servers, how can we ensure each application server gets an even load? The load balancer handles this nicely.

Step 4 - With the business continuing to grow, the database might become the bottleneck. To mitigate this, we separate reads and writes in a way that frequent read queries go to read replicas. With this setup, the throughput for the database writes can be greatly increased.

Step 5 - Suppose the business continues to grow. One single database cannot handle the load on both the inventory table and user table. We have a few options:

1. Vertical partition. Adding more power (CPU, RAM, etc.) to the database server. It has a hard limit.
2. Horizontal partition by adding more database servers.
3. Adding a caching layer to offload read requests.

Step 6 - Now we can modularize the functions into different services. The architecture becomes service-oriented / microservice.

Question: what else do we need to support an e-commerce website at Amazon’s scale?

DevOps Books

Some **DevOps** books I find enlightening:

🔹 Accelerate - presents both the findings and the science behind measuring software delivery performance.

🔹 Continuous Delivery - introduces automated architecture management and data migration. It also pointed out key problems and optimal solutions in each area.

🔹 Site Reliability Engineering - famous Google SRE book. It explains the whole life cycle of Google's development, deployment, and monitoring, and how to manage the world's biggest software systems.

🔹 Effective DevOps - provides effective ways to improve team coordination.

🔹 The Phoenix Project - a classic novel about effectiveness and communications. IT work is like manufacturing plant work, and a system must be established to streamline the workflow. Very interesting read!

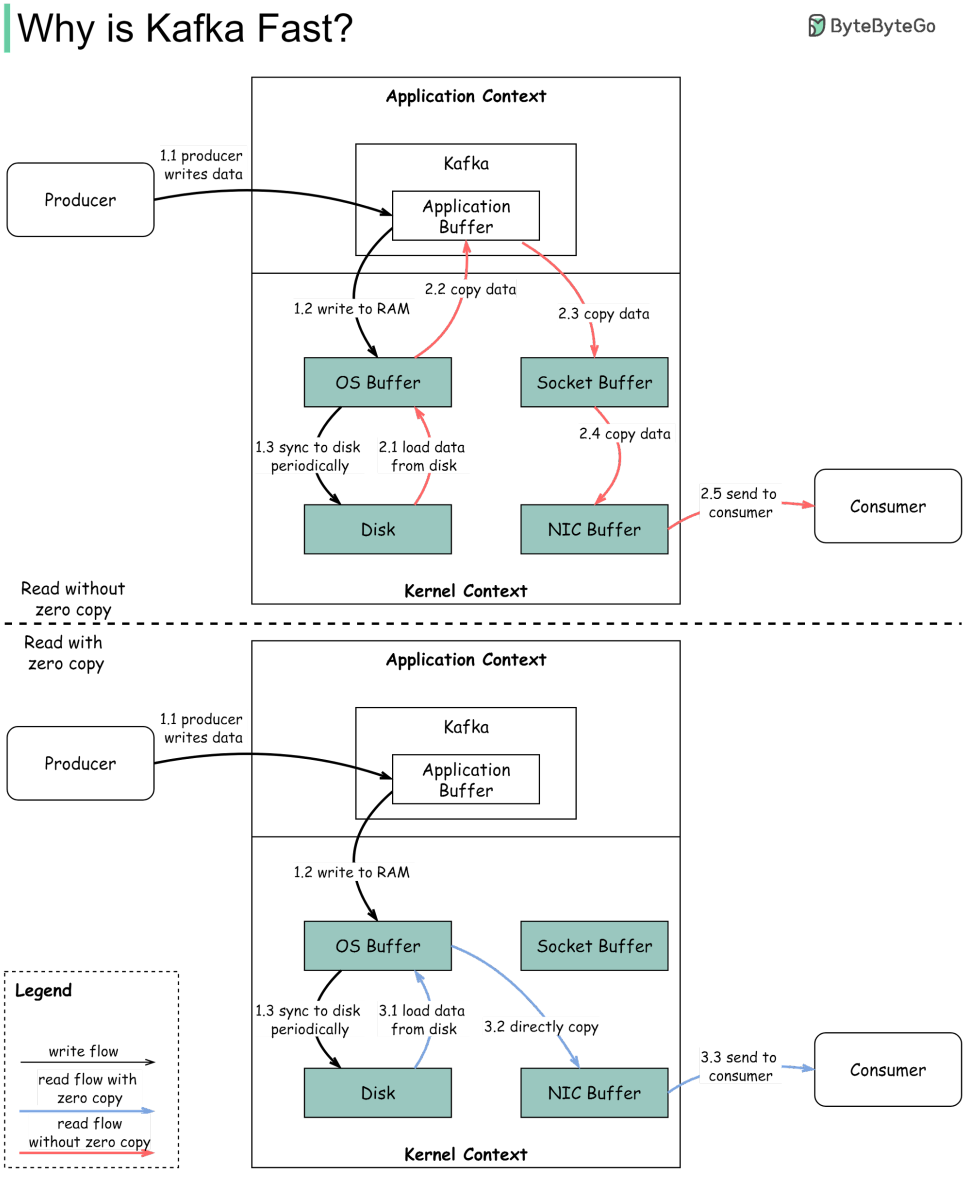
🔹 The DevOps Handbook - introduces product development, quality assurance, IT operations, and information security.

What’s your favorite dev-ops book?

Why is Kafka fast?

Kafka achieves low latency message delivery through Sequential I/O and Zero Copy Principle. The same techniques are commonly used in many other messaging/streaming platforms.

The diagram below illustrates how the data is transmitted between producer and consumer, and what zero-copy means.



- Step 1.1 - 1.3: Producer writes data to the disk

🔹 Step 2: Consumer reads data without zero-copy
2.1: The data is loaded from disk to OS cache
2.2 The data is copied from OS cache to Kafka application
2.3 Kafka application copies the data into the socket buffer
2.4 The data is copied from socket buffer to network card
2.5 The network card sends data out to the consumer

🔹 Step 3: Consumer reads data with zero-copy
3.1: The data is loaded from disk to OS cache
3.2 OS cache directly copies the data to the network card via sendfile() command
3.3 The network card sends data out to the consumer

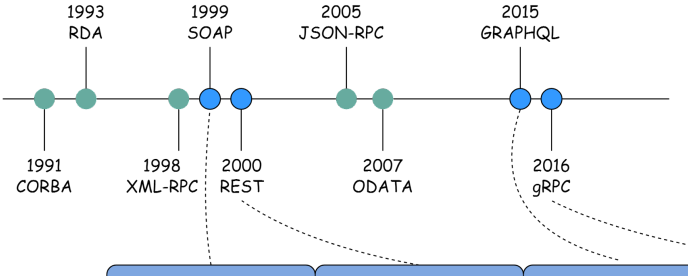
Zero copy is a shortcut to save the multiple data copies between application context and kernel context. This approach brings down the time by approximately 65%.

—

SOAP vs REST vs GraphQL vs RPC.

The diagram below illustrates the API timeline and API styles comparison.

> API Architectural Styles Comparison



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

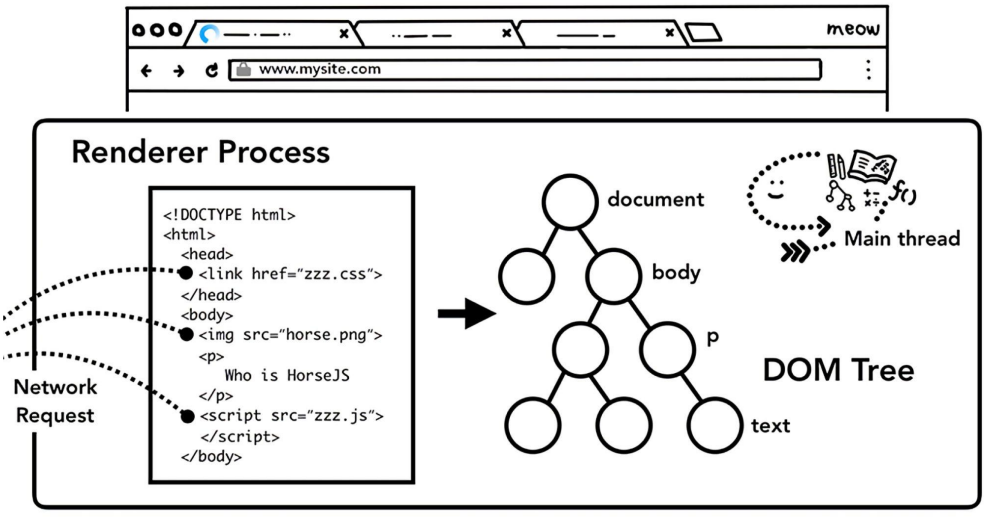
Over time, different API architectural styles are released. Each of them has its own patterns of standardizing data exchange.

You can check out the use cases of each style in the diagram.

Source: https://lnkd.in/gFgi33RY I combined a few diagrams together. The credit all goes to AltexSoft.

—

How do modern browsers work?



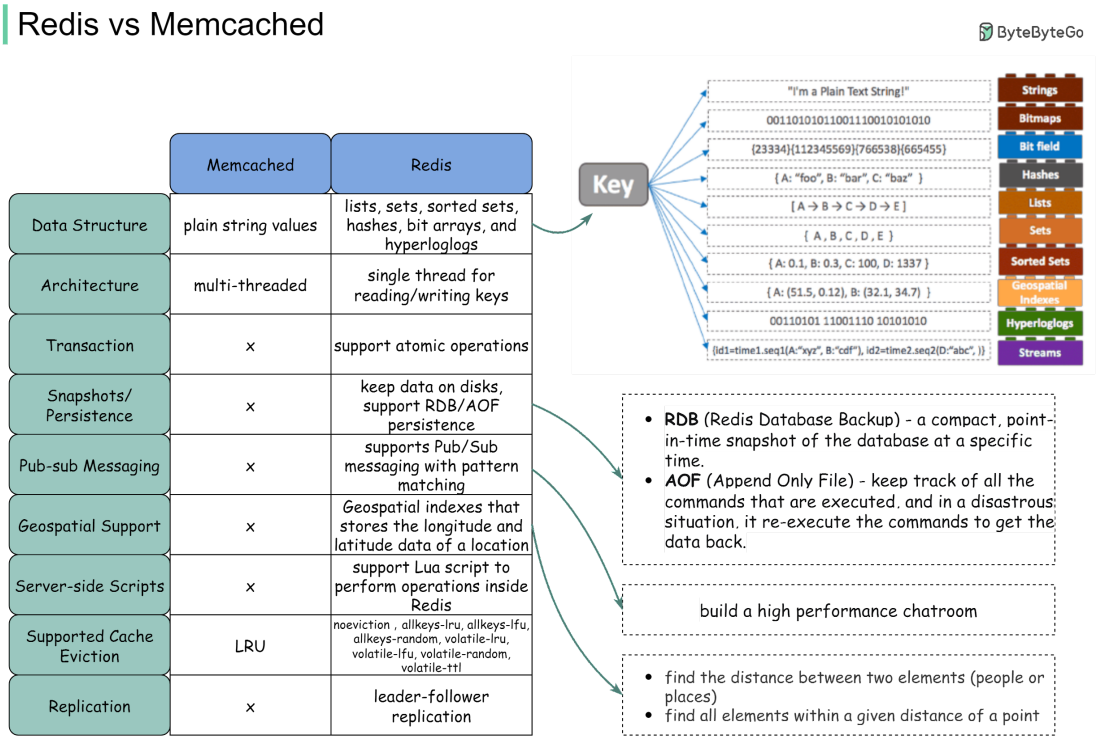
Google published a series of articles about "Inside look at modern web browser". It's a great read.

Links:
https://developer.chrome.com/blog/inside-browser-part1/
https://developer.chrome.com/blog/inside-browser-part2/
https://developer.chrome.com/blog/inside-browser-part3/
https://developer.chrome.com/blog/inside-browser-part4/

Redis vs Memcached

The diagram below illustrates the key differences.

Redis vs Memcached



The advantages on data structures make Redis a good choice for:

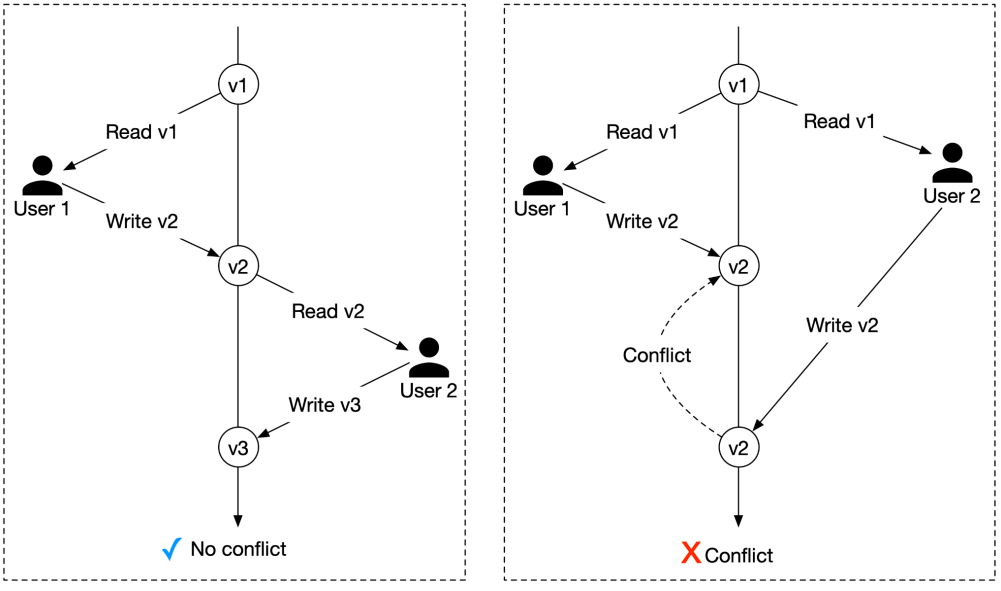
- Recording the number of clicks and comments for each post (hash)
- Sorting the commented user list and deduping the users (zset)
- Caching user behavior history and filtering malicious behaviors (zset, hash)
- Storing boolean information of extremely large data into small space. For example, login status, membership status. (bitmap)

Optimistic locking

Optimistic locking, also referred to as optimistic concurrency control, allows multiple concurrent users to attempt to update the same resource.

There are two common ways to implement optimistic locking: version number and timestamp. Version number is generally considered to be a better option because the server clock can be inaccurate over time. We explain how optimistic locking works with version number.

The diagram below shows a successful case and a failure case.



1. A new column called “version” is added to the database table.

2. Before a user modifies a database row, the application reads the version number of the row.

3. When the user updates the row, the application increases the version number by 1 and writes it back to the database.

4. A database validation check is put in place; the next version number should exceed the current version number by 1. The transaction aborts if the validation fails and the user tries again from step 2.

Optimistic locking is usually faster than pessimistic locking because we do not lock the database. However, the performance of optimistic locking drops dramatically when concurrency is high.

To understand why, consider the case when many clients try to reserve a hotel room at the same time. Because there is no limit on how many clients can read the available room count, all of them read back the same available room count and the current version number. When different clients make reservations and write back the results to the database, only one of them will succeed, and the rest of the clients receive a version check failure message. These clients have to retry. In the subsequent round of retries, there is only one successful client again, and the rest have to retry. Although the end result is correct, repeated retries cause a very unpleasant user experience.

Question: what are the possible ways of solving race conditions?

Tradeoff between latency and consistency

Understanding the **tradeoffs** is very important not only in system design interviews but also designing real-world systems. When we talk about data replication, there is a fundamental tradeoff between **latency** and **consistency**. It is illustrated by the diagram below.

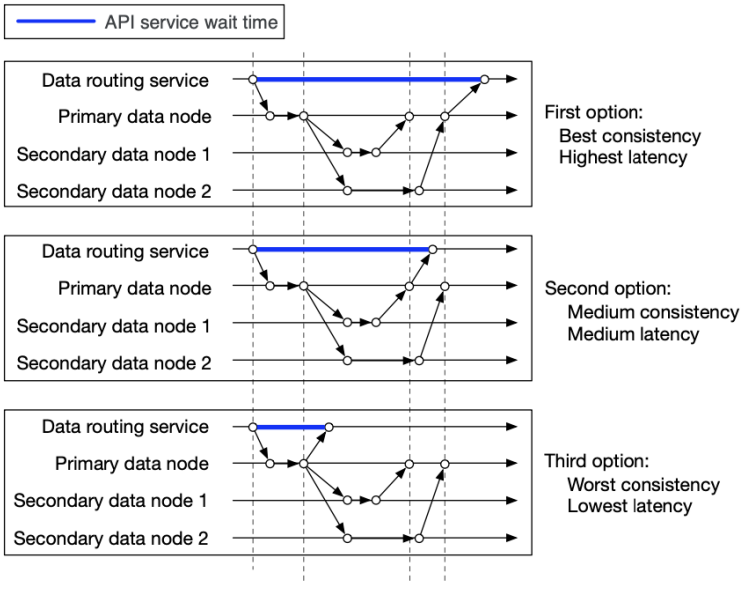


Figure 9.11: Trade-off between consistency and latency

1. Data is considered as successfully saved after all three nodes store the data. This approach has the best consistency but the highest latency.
2. Data is considered as successfully saved after the primary and one of the secondaries store the data. This approach has a medium consistency and medium latency.
3. Data is considered as successfully saved after the primary persists the data. This approach has the worst consistency but the lowest latency.

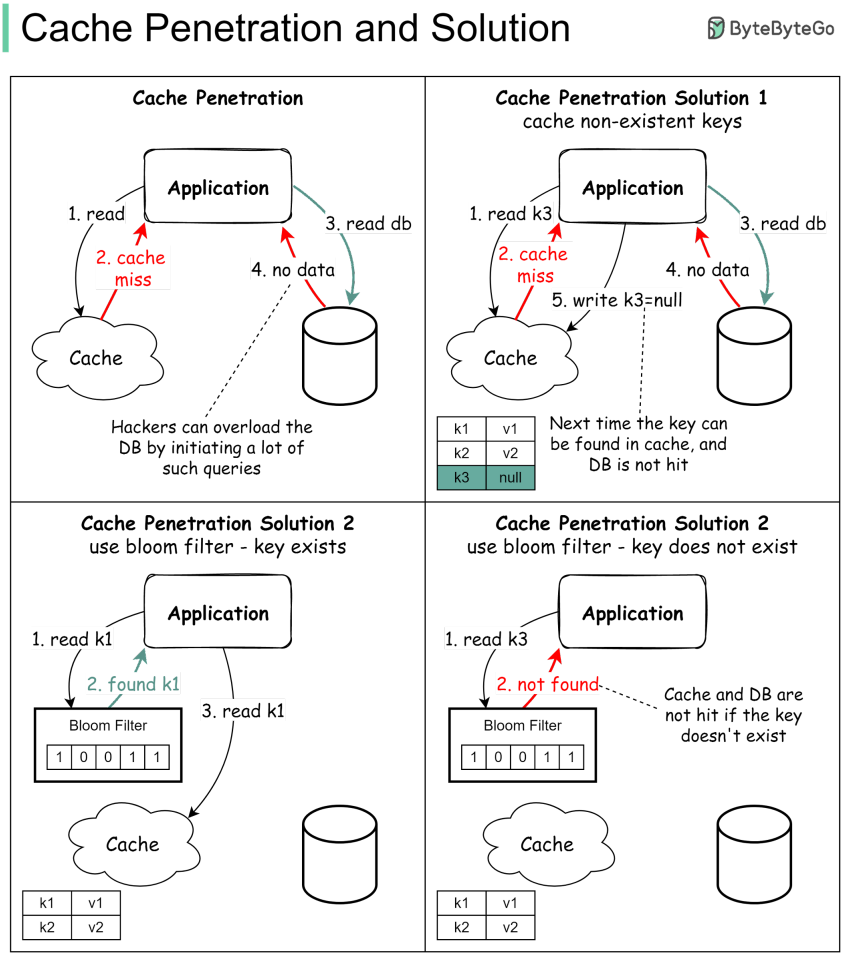
Both 2 and 3 are forms of eventual consistency.

Cache miss attack

Caching is awesome but it doesn’t come without a cost, just like many things in life.

One of the issues is **Cache Miss Attack**. Correct me if this is not the right term. It refers to the scenario where data to fetch doesn't exist in the database and the data isn’t cached either. So every request hits the database eventually, defeating the purpose of using a cache. If a malicious user initiates lots of queries with such keys, the database can easily be overloaded.

The diagram below illustrates the process.



Two approaches are commonly used to solve this problem:

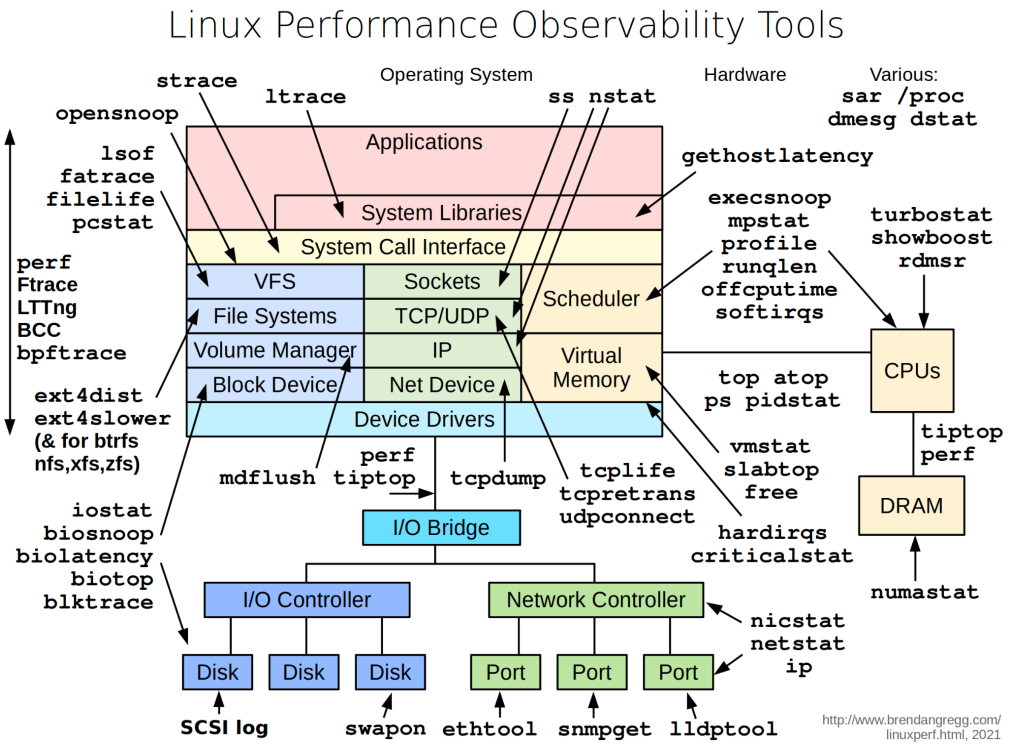
🔹 Cache keys with null value. Set a short TTL (Time to Live) for keys with null value.

🔹 Using Bloom filter. A Bloom filter is a data structure that can rapidly tell us whether an element is present in a set or not. If the key exists, the request first goes to the cache and then queries the database if needed. If the key doesn't exist in the data set, it means the key doesn’t exist in the cache/database. In this case, the query will not hit the cache or database layer.

—

How to diagnose a mysterious process that's taking too much CPU, memory, IO, etc?

The diagram below illustrates helpful tools in a Linux system.



- 'vmstat' - reports information about processes, memory, paging, block IO, traps, and CPU activity.

- 'iostat' - reports CPU and input/output statistics of the system.

- 'netstat' - displays statistical data related to IP, TCP, UDP, and ICMP protocols.

- 'lsof' - lists open files of the current system.

- 'pidstat' - monitors the utilization of system resources by all or specified processes, including CPU, memory, device IO, task switching, threads, etc.

What are the top cache strategies?

Read data from the system:

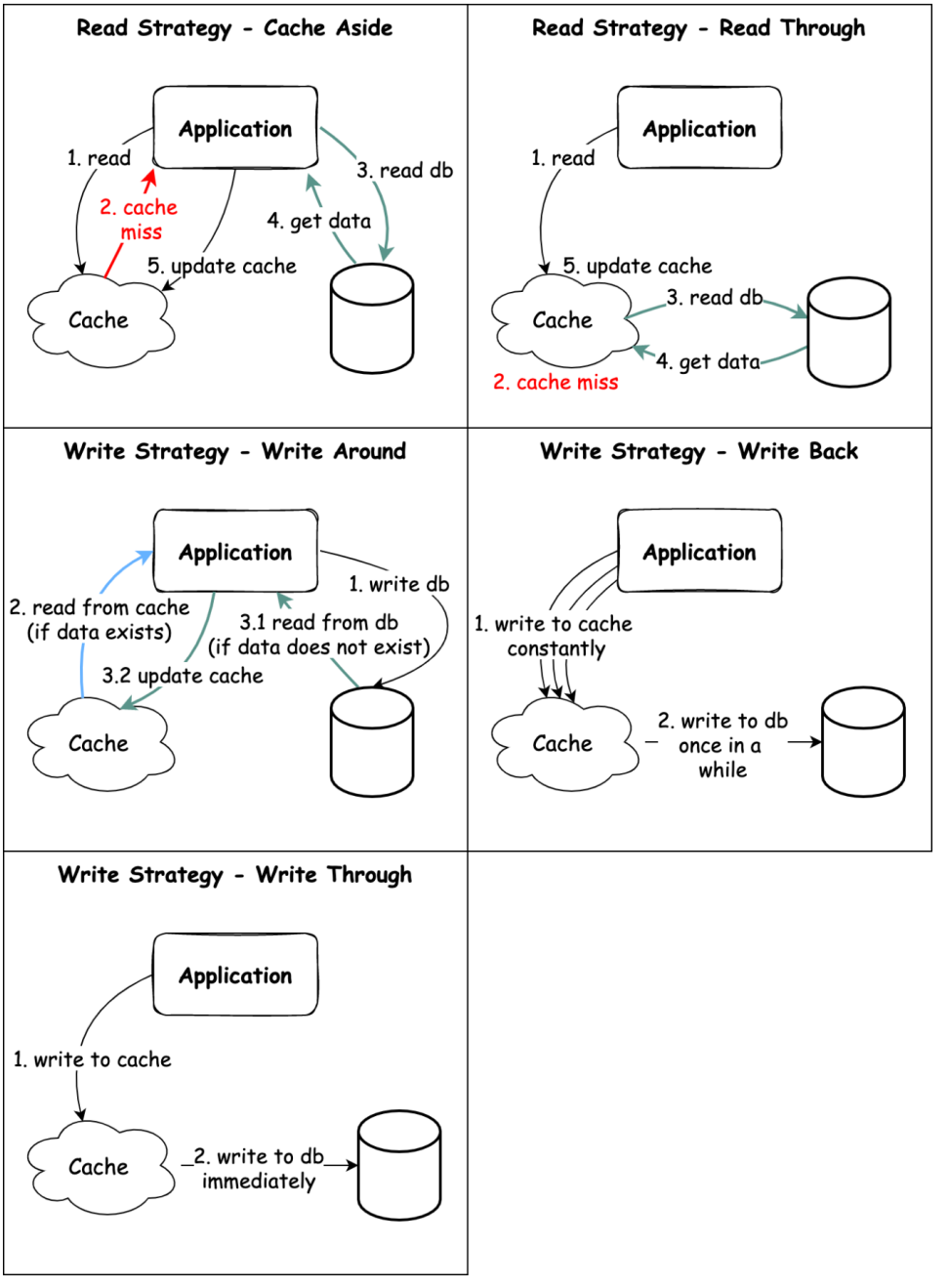
- Cache aside
- Read through

Write data to the system:

- Write around
- Write back
- Write through

The diagram below illustrates how those 5 strategies work. Some of the caching strategies can be used together.

Top caching strategies



I left out a lot of details as that will make the post very long. Feel free to leave a comment so we can learn from each other.

Question: What are the pros and cons of each caching strategy? How to choose the right one to use?

—

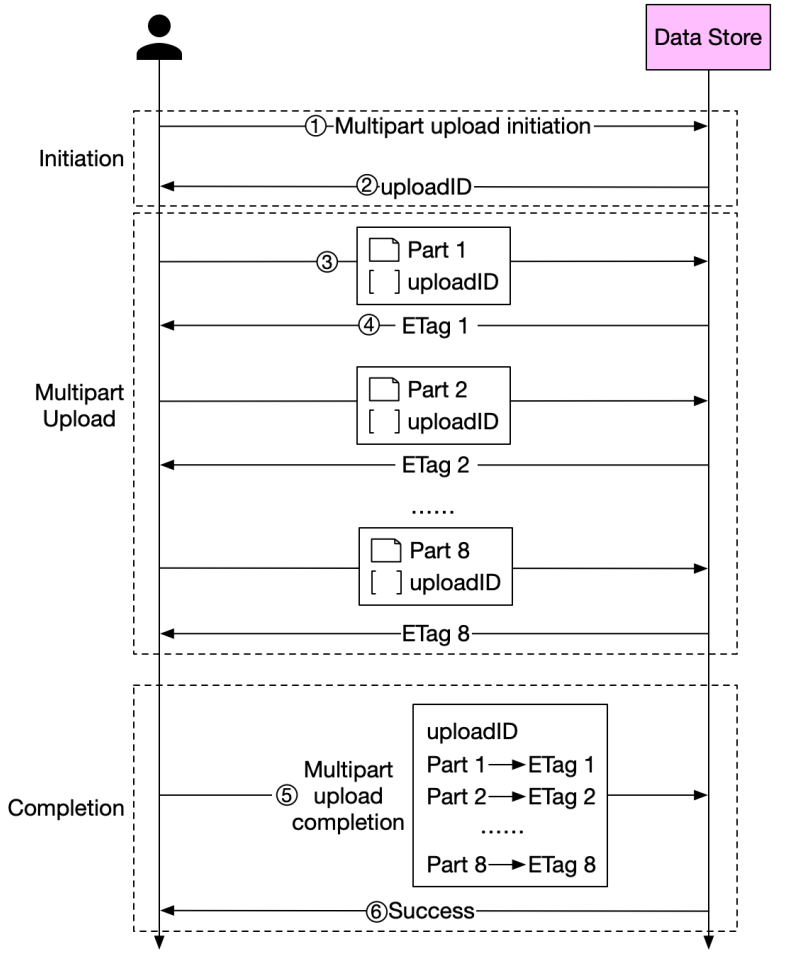
Check out our bestselling system design books.
Paperback: Amazon Digital: ByteByteGo.

Upload large files

How can we optimize performance when we **upload large files** to object storage service such as S3?

Before we answer this question, let's take a look at why we need to optimize this process. Some files might be larger than a few GBs. It is possible to upload such a large object file directly, but it could take a long time. If the network connection fails in the middle of the upload, we have to start over. A better solution is to slice a large object into smaller parts and upload them independently. After all the parts are uploaded, the object store re-assembles the object from the parts. This process is called **multipart upload**.

The diagram below illustrates how multipart upload works:



1. The client calls the object storage to initiate a multipart upload.

2. The data store returns an uploadID, which uniquely identifies the upload.

3. The client splits the large file into small objects and starts uploading. Let's assume the size of the file is 1.6GB and the client splits it into 8 parts, so each part is 200 MB in size. The client uploads the first part to the data store together with the uploadID it received in step 2.

4. When a part is uploaded, the data store returns an ETag, which is essentially the md5 checksum of that part. It is used to verify multipart uploads.

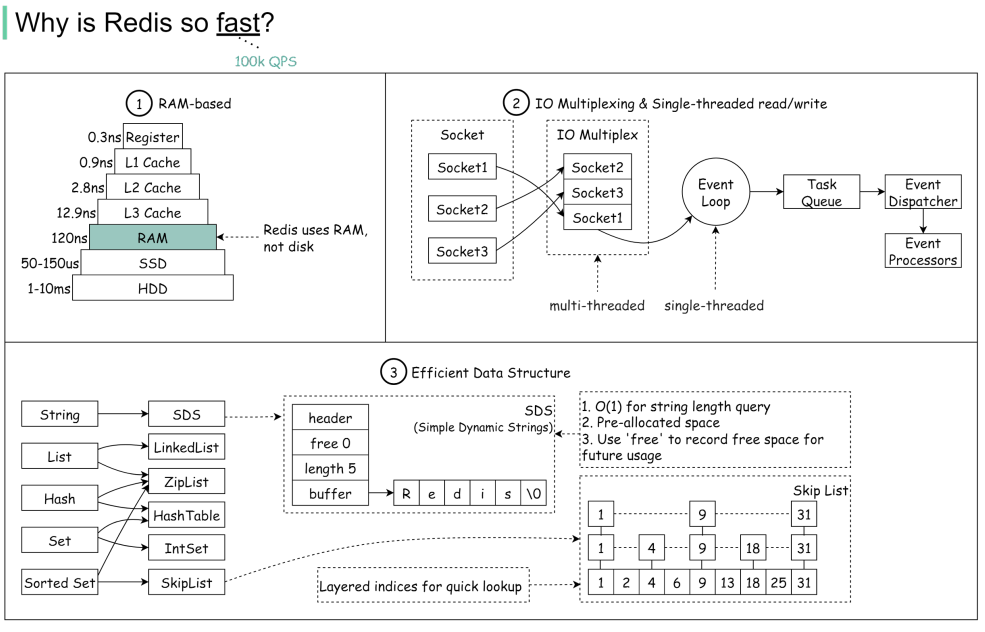
5. After all parts are uploaded, the client sends a complete multipart upload request, which includes the uploadID, part numbers, and ETags.

6. The data store reassembles the object from its parts based on the part number. Since the object is really large, this process may take a few minutes. After reassembly is complete, it returns a success message to the client.

—

Why is Redis so Fast?

There are 3 main reasons as shown in the diagram below.



1. Redis is a RAM-based database. RAM access is at least 1000 times faster than random disk access.

2. Redis leverages IO multiplexing and single-threaded execution loop for execution efficiency.

3. Redis leverages several efficient lower-level data structures.

Question: Another popular in-memory store is Memcached. Do you know the differences between Redis and Memcached?

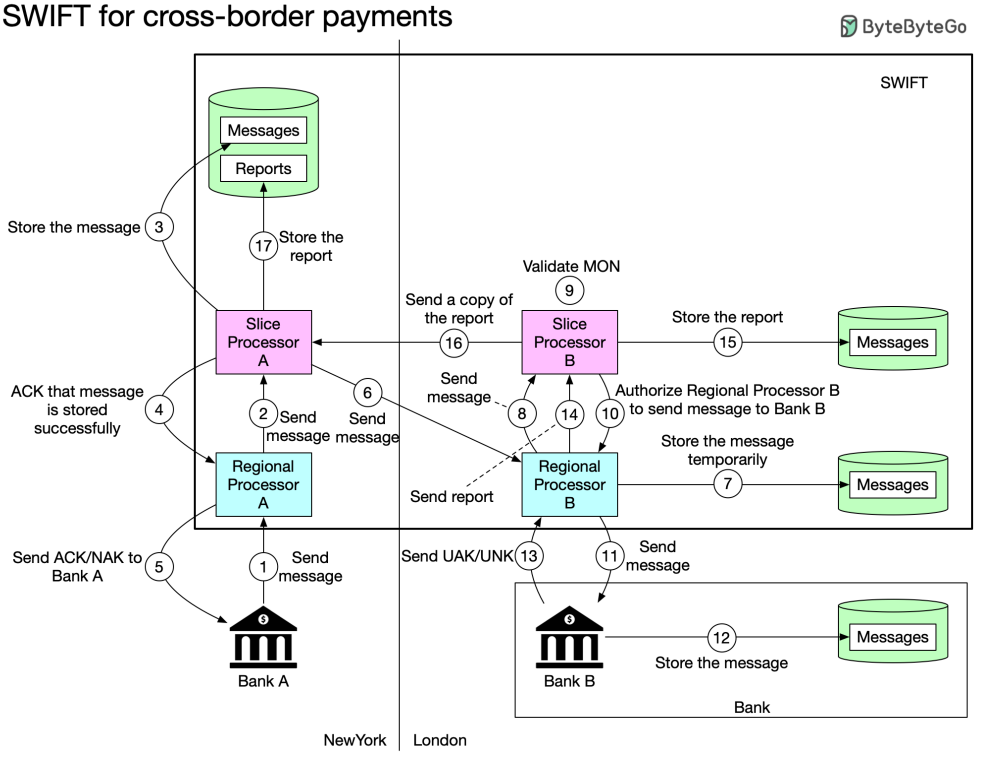
You might have noticed the style of this diagram is different from my previous posts. Please let me know which one you prefer.

—

SWIFT payment network

You probably heard about **SWIFT**. What is SWIFT? What role does it play in cross-border payments? You can find answers to those questions in this post.

SWIFT for cross-border payments



The Society for Worldwide Interbank Financial Telecommunication (SWIFT) is the main secure **messaging system** that links the world's banks.

The Belgium-based system is run by its member banks and handles millions of payment messages per day. The diagram below illustrates how payment messages are transmitted from Bank A (in New York) to Bank B (in London).

Step 1: Bank A sends a message with transfer details to Regional Processor A in New York. The destination is Bank B.

Step 2: Regional processor validates the format and sends it to Slice Processor A. The Regional Processor is responsible for input message validation and output message queuing. The Slice Processor is responsible for storing and routing messages safely.

Step 3: Slice Processor A stores the message.

Step 4: Slice Processor A informs Regional Processor A the message is stored.

Step 5: Regional Processor A sends ACK/NAK to Bank A. ACK means a message will be sent to Bank B. NAK means the message will NOT be sent to Bank B.

Step 6: Slice Processor A sends the message to Regional Processor B in London.

Step 7: Regional Processor B stores the message temporarily.

Step 8: Regional Processor B assigns a unique ID MON (Message Output Number) to the message and sends it to Slice Processor B

Step 9: Slice Processor B validates MON.

Step 10: Slice Processor B authorizes Regional Processor B to send the message to Bank B.

Step 11: Regional Processor B sends the message to Bank B.

Step 12: Bank B receives the message and stores it.

Step 13: Bank B sends UAK/UNK to Regional Processor B. UAK (user positive acknowledgment) means Bank B received the message without error; UNK (user negative acknowledgment) means Bank B received checksum failure.

Step 14: Regional Processor B creates a report based on Bank B’s response, and sends it to Slice Processor B.

Step 15: Slice Processor B stores the report.

Step 16 - 17: Slice Processor B sends a copy of the report to Slice Processor A. Slice Processor A stores the report.

—

Check out our bestselling system design books.
Paperback: Amazon Digital: ByteByteGo.

At-most once, at-least once, and exactly once

In modern architecture, systems are broken up into small and independent building blocks with well-defined interfaces between them. Message queues provide communication and coordination for those building blocks. Today, let's discuss different delivery semantics: at-most once, at-least once, and exactly once.

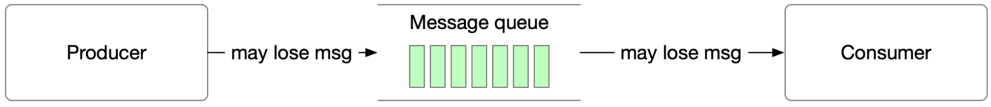


Figure 1 At-most once

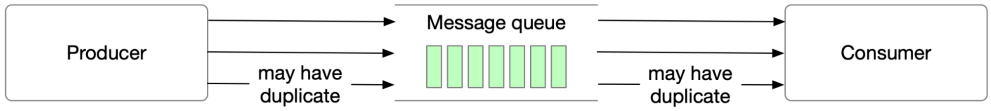


Figure 2 At-least once

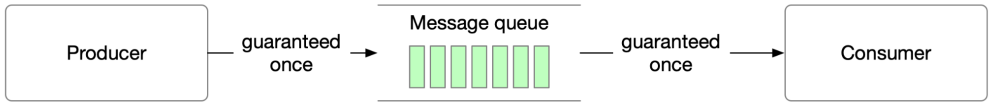


Figure 3 Exactly-once

At-most once

As the name suggests, at-most once means a message will be delivered not more than once. Messages may be lost but are not redelivered. This is how at-most once delivery works at the high level.

Use cases: It is suitable for use cases like monitoring metrics, where a small amount of data loss is acceptable.

At-least once

With this data delivery semantic, it's acceptable to deliver a message more than once, but no message should be lost.

Use cases: With at-least once, messages won't be lost but the same message might be delivered multiple times. While not ideal from a user perspective, at-least once delivery semantics are usually good enough for use cases where data duplication is not a big issue or deduplication

is possible on the consumer side. For example, with a unique key in each message, a message can be rejected when writing duplicate data to the database.

Exactly once

Exactly once is the most difficult delivery semantic to implement. It is friendly to users, but it has a high cost for the system's performance and complexity.

Use cases: Financial-related use cases (payment, trading, accounting, etc.). Exactly once is especially important when duplication is not acceptable and the downstream service or third party doesn't support idempotency.

Question: what is the difference between message queues vs event streaming platforms such as Kafka, Apache Pulsar, etc?

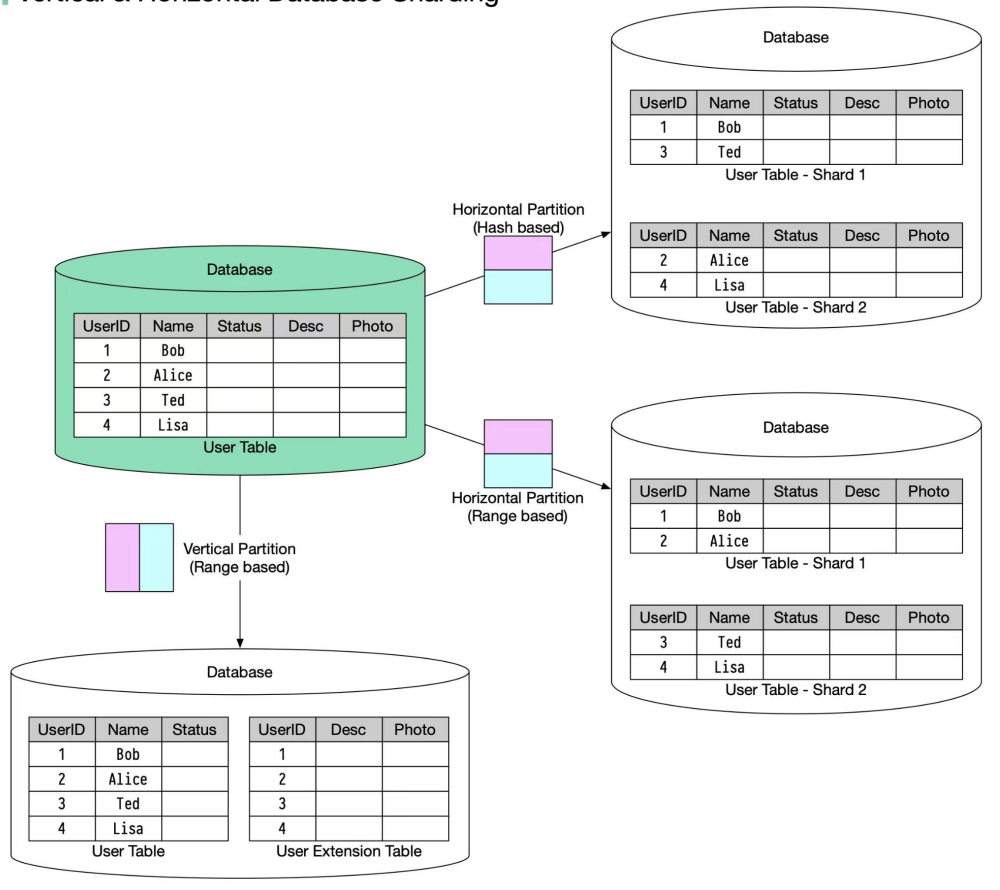
Vertical partitioning and Horizontal partitioning

In many large-scale applications, data is divided into partitions that can be accessed separately. There are two typical strategies for partitioning data.

- Vertical partitioning: it means some columns are moved to new tables. Each table contains the same number of rows but fewer columns (see diagram below).

- Horizontal partitioning (often called sharding): it divides a table into multiple smaller tables. Each table is a separate data store, and it contains the same number of columns, but fewer rows (see diagram below).

Vertical & Horizontal Database Sharding



Horizontal partitioning is widely used so let’s take a closer look.

Routing algorithm
Routing algorithm decides which partition (shard) stores the data.

- Range-based sharding. This algorithm uses ordered columns, such as integers, longs, timestamps, to separate the rows. For example, the diagram below uses the User ID column for range partition: User IDs 1 and 2 are in shard 1, User IDs 3 and 4 are in shard 2.

- Hash-based sharding. This algorithm applies a hash function to one column or several columns to decide which row goes to which table. For example, the diagram below uses **User ID mod** 2 as a hash function. User IDs 1 and 3 are in shard 1, User IDs 2 and 4 are in shard 2.

Benefits
- Facilitate horizontal scaling. Sharding facilitates the possibility of adding more machines to spread out the load.

- Shorten response time. By sharding one table into multiple tables, queries go over fewer rows, and results are returned much more quickly.

Drawbacks
- The order by the operation is more complicated. Usually, we need to fetch data from different shards and sort the data in the application's code.

- Uneven distribution. Some shards may contain more data than others (this is also called the hotspot).

This topic is very big and I’m sure I missed a lot of important details. What else do you think is important for data partitioning?

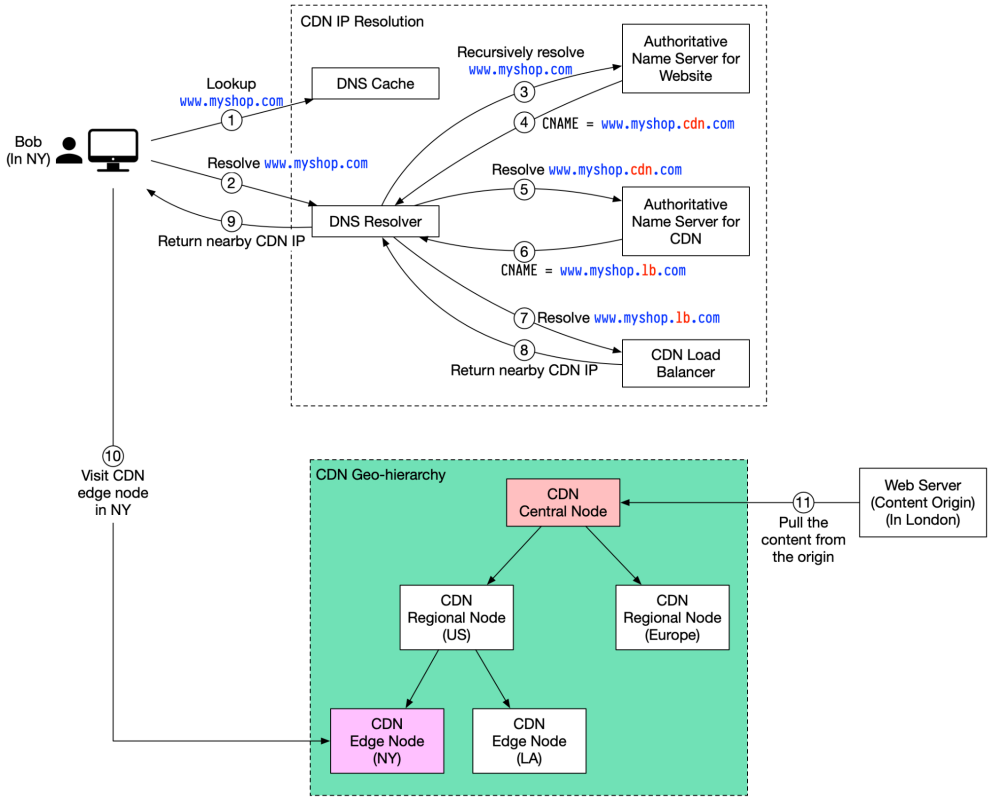
CDN

A content delivery network (CDN) refers to a geographically distributed servers (also called edge servers) which provide fast delivery of static and dynamic content. Let's take a look at how it works.

Suppose Bob who lives in New York wants to visit an eCommerce website that is deployed in London. If the request goes to servers located in London, the response will be quite slow. So we deploy CDN servers close to where Bob lives, and the content will be loaded from the nearby CDN server.

The diagram below illustrates the process:

How does CDN work



1. Bob types in www.myshop.com in the browser. The browser looks up the domain name in the local DNS cache.

2. If the domain name does not exist in the local DNS cache, the browser goes to the DNS resolver to resolve the name. The DNS resolver usually sits in the Internet Service Provider (ISP).

3. The DNS resolver recursively resolves the domain name (see my previous post for details). Finally, it asks the authoritative name server to resolve the domain name.

4. If we don’t use CDN, the authoritative name server returns the IP address for www.myshop.com. But with CDN, the authoritative name server has an alias pointing to www.myshop.cdn.com (the domain name of the CDN server).

5. The DNS resolver asks the authoritative name server to resolve www.myshop.cdn.com.

6. The authoritative name server returns the domain name for the load balancer of CDN www.myshop.lb.com.

7. The DNS resolver asks the CDN load balancer to resolve www.myshop.lb.com. The load balancer chooses an optimal CDN edge server based on the user’s IP address, user’s ISP, the content requested, and the server load.

8. The CDN load balancer returns the CDN edge server’s IP address for www.myshop.lb.com.

9. Now we finally get the actual IP address to visit. The DNS resolver returns the IP address to the browser.

10. The browser visits the CDN edge server to load the content. There are two types of contents cached on the CDN servers: static contents and dynamic contents. The former contains static pages, pictures, and videos; the latter one includes results of edge computing.

11. If the edge CDN server cache doesn't contain the content, it goes upward to the regional CDN server. If the content is still not found, it will go upward to the central CDN server, or even go to the origin - the

London web server. This is called the CDN distribution network, where the servers are deployed geographically.

Over to you: How do you prevent videos cached on CDN from being pirated?

Erasure coding

A really cool technique that's commonly used in object storage such as S3 to improve durability is called **Erasure Coding**. Let's take a look at how it works.

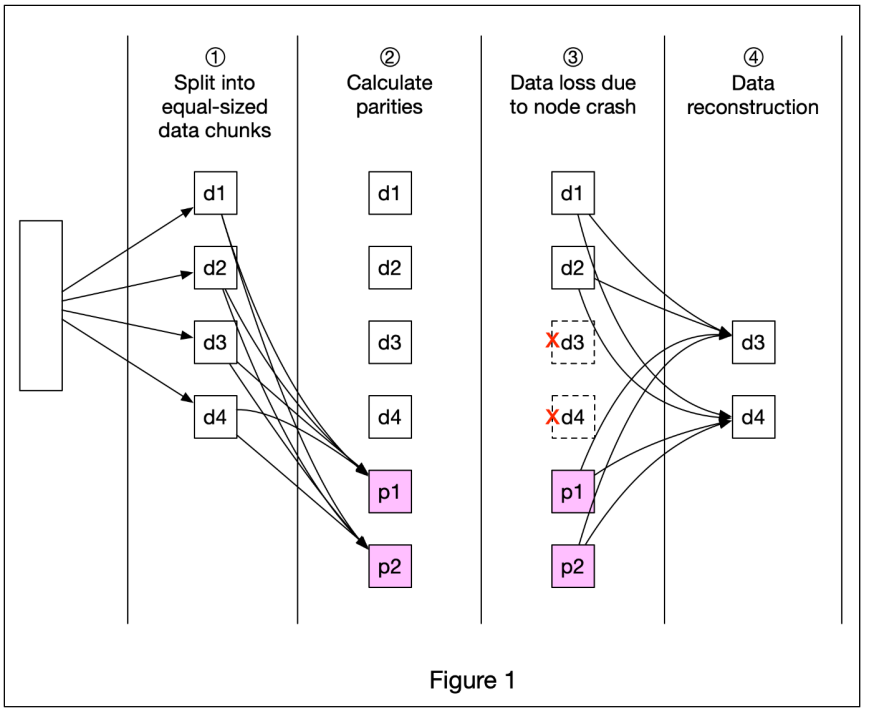


Figure 1

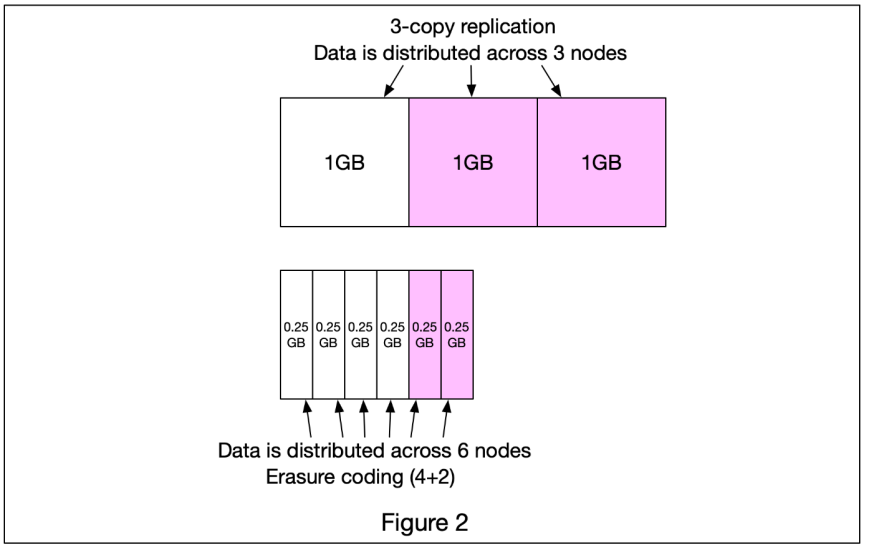


Figure 2

Erasure coding deals with data durability differently from replication. It chunks data into smaller pieces (placed on different servers) and creates parities for redundancy. In the event of failures, we can use chunk data and parities to reconstruct the data. Let's take a look at a concrete example (4 + 2 erasure coding) as shown in Figure 1.

1 Data is broken up into four even-sized data chunks d1, d2, d3, and d4.

2 The mathematical formula is used to calculate the parities p1 and p2. To give a much simplified example, p1 = d1 + 2*d2 - d3 + 4*d4 and p2 = -d1 + 5*d2 + d3 - 3*d4.

3 Data d3 and d4 are lost due to node crashes.

4 The mathematical formula is used to reconstruct lost data d3 and d4, using the known values of d1, d2, p1, and p2.

How much extra space does erasure coding need? For every two chunks of data, we need one parity block, so the storage overhead is 50% (Figure 2). While in 3-copy replication, the storage overhead is 200% (Figure 2).

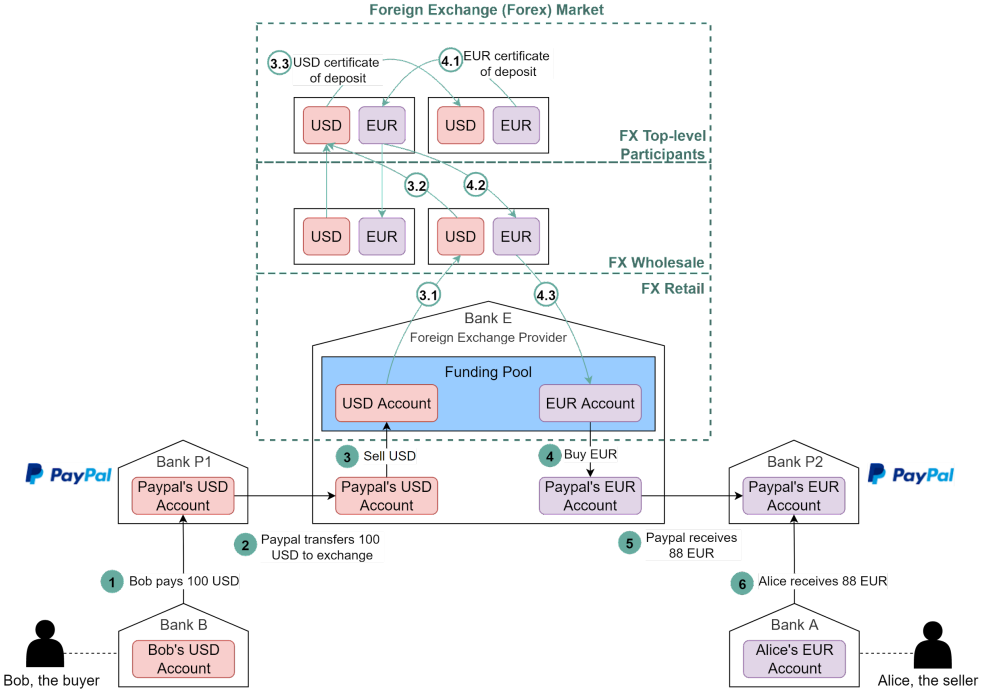
Does erasure coding increase data durability? Let's assume a node has a 0.81% annual failure rate. According to the calculation done by Backblaze, erasure coding can achieve 11 nines durability vs 3-copy replication can achieve 6 nines durability.

What other techniques do you think are important to improve the scalability and durability of an object store such as S3?

Foreign exchange in payment

Have you wondered what happens under the hood when you pay with USD online and the seller from Europe receives EUR (euro)? This process is called foreign exchange.

Foreign Exchange in Payments



Suppose Bob (the buyer) needs to pay 100 USD to Alice (the seller), and Alice can only receive EUR. The diagram below illustrates the process.

1. Bob sends 100 USD via a third-party payment provider. In our example, it is Paypal. The money is transferred from Bob's bank account (Bank B) to Paypal's account in Bank P1.

2. Paypal needs to convert USD to EUR. It leverages the foreign exchange provider (Bank E). Paypal sends 100 USD to its USD account in Bank E.

3. 100 USD is sold to Bank E's funding pool.

4. Bank E's funding pool provides 88 EUR in exchange for 100 USD. The money is put into Paypal's EUR account in Bank E.

5. Paypal's EUR account in Bank P2 receives 88 EUR.

6. 88 EUR is paid to Alice's EUR account in Bank A.

Now let's take a close look at the foreign exchange (forex) market. It has 3 layers:

- Retail market. Funding pools are parts of the retail market. To improve efficiency, Paypal usually buys a certain amount of foreign currencies in advance.
- Wholesale market. The wholesale business is composed of investment banks, commercial banks, and foreign exchange providers. It usually handles accumulated orders from the retail market.
- Top-level participants. They are multinational commercial banks that hold a large number of certificates of deposit from different countries. They exchange these certificates for foreign exchange trading.

When Bank E's funding pool needs more EUR, it goes upward to the wholesale market to sell USD and buy EUR. When the wholesale market accumulates enough orders, it goes upward to top-level participants. Steps 3.1-3.3 and 4.1-4.3 explain how it works.

If you have any questions, please leave a comment.

What foreign currency did you find difficult to exchange? And what company have you used for foreign currency exchange?

Interview Question: Design S3

What happens when you upload a file to Amazon S3? Let's design an S3 like object storage system.

Upload a File to S3

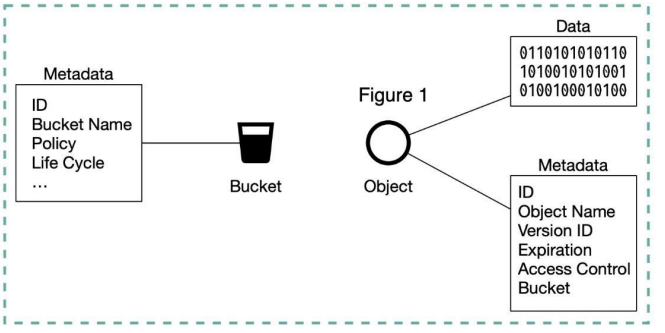


Figure 1

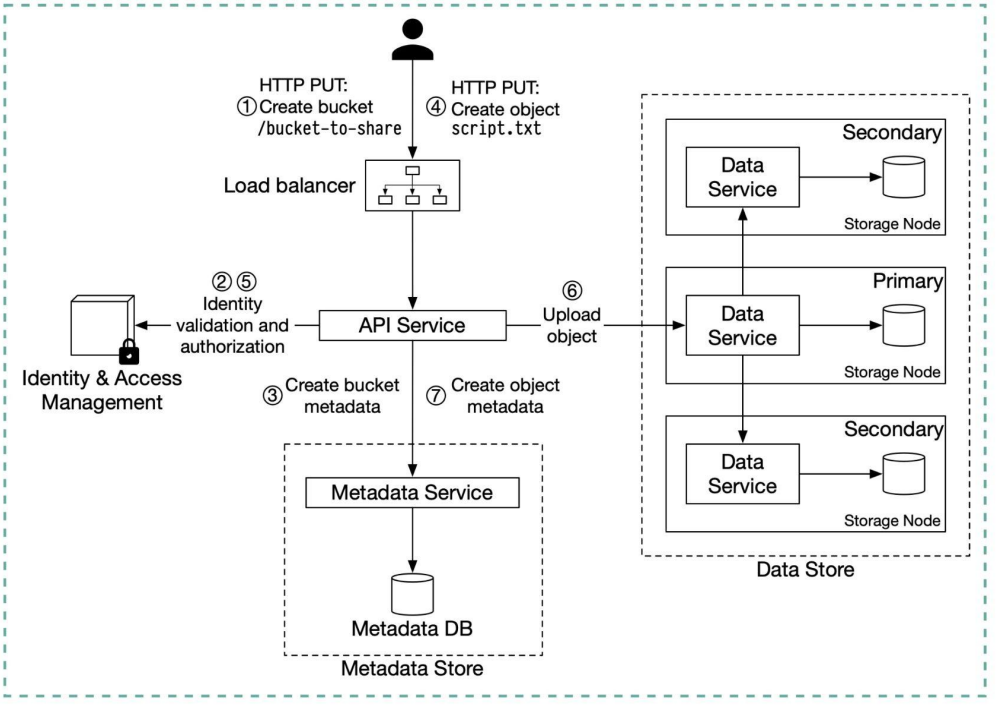


Figure 2

Before we dive into the design, let's define some terms.

Bucket. A logical container for objects. The bucket name is globally unique. To upload data to S3, we must first create a bucket.

Object. An object is an individual piece of data we store in a bucket. It contains object data (also called payload) and metadata. Object data can be any sequence of bytes we want to store. The metadata is a set of name-value pairs that describe the object.

An S3 object consists of (Figure 1):

- Metadata. It is mutable and contains attributes such as ID, bucket name, object name, etc.
- Object data. It is immutable and contains the actual data.

In S3, an object resides in a bucket. The path looks like this: /bucket-to-share/script.txt. The bucket only has metadata. The object has metadata and the actual data.

The diagram below (Figure 2) illustrates how file uploading works. In this example, we first create a bucket named “bucket-to-share” and then upload a file named “script.txt” to the bucket.

1. The client sends an HTTP PUT request to create a bucket named “bucket-to-share.” The request is forwarded to the API service.

2. The API service calls the Identity and Access Management (IAM) to ensure the user is authorized and has WRITE permission.

3. The API service calls the metadata store to create an entry with the bucket info in the metadata database. Once the entry is created, a success message is returned to the client.

4. After the bucket is created, the client sends an HTTP PUT request to create an object named “script.txt”.

5. The API service verifies the user’s identity and ensures the user has WRITE permission on the bucket.

6. Once validation succeeds, the API service sends the object data in the HTTP PUT payload to the data store. The data store persists the payload as an object and returns the UUID of the object.

7. The API service calls the metadata store to create a new entry in the metadata database. It contains important metadata such as the object_id (UUID), bucket_id (which bucket the object belongs to), object_name, etc.

Block storage, file storage and object storage

Yesterday, I posted the definitions of block storage, file storage, and object storage. Let's continue the discussion and compare those 3 options.

	Block storage	File storage	Object storage
Mutable Content	Y	Y	N (object versioning is supported, in-place update is not)
Cost	High	Medium to high	Low
Performance	Medium to high, very high	Medium to high	Low to medium
Consistency	Strong consistency	Strong consistency	Strong consistency
Data access	SAS/iSCSI/FC	Standard file access, CIFS/SMB, and NFS	RESTful API
Scalability	Medium scalability	High scalability	Vast scalability
Good for	Virtual machines (VM), high-performance applications like database	General-purpose file system access	Binary data, unstructured data

Table 1 Storage options

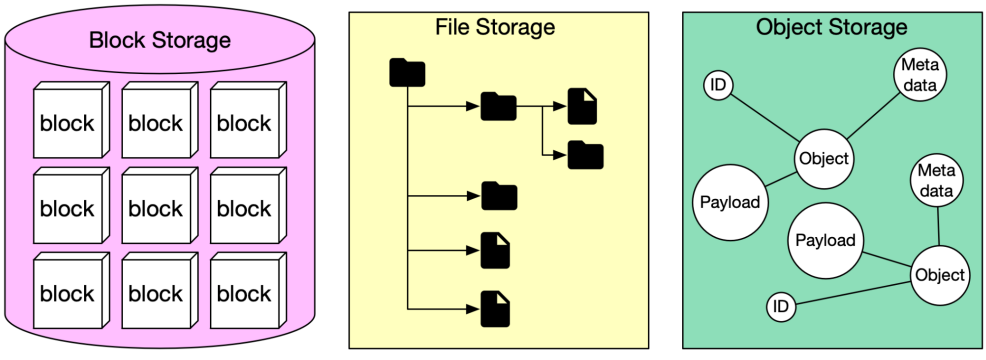
Block storage, file storage and object storage

In this post, let's review the storage systems in general.

Storage systems fall into three broad categories:

- Block storage
- File storage
- Object storage

The diagram below illustrates the comparison of different storage systems.



Block storage

Block storage came first, in the 1960s. Common storage devices like hard disk drives (HDD) and solid-state drives (SSD) that are physically attached to servers are all considered as block storage.

Block storage presents the raw blocks to the server as a volume. This is the most flexible and versatile form of storage. The server can format the raw blocks and use them as a file system, or it can hand control of those blocks to an application. Some applications like a database or a virtual machine engine manage these blocks directly in order to squeeze every drop of performance out of them.

Block storage is not limited to physically attached storage. Block storage could be connected to a server over a high-speed network or over industry-standard connectivity protocols like Fibre Channel (FC)

and iSCSI. Conceptually, the network-attached block storage still presents raw blocks. To the servers, it works the same as physically attached block storage. Whether to a network or physically attached, block storage is fully owned by a single server. It is not a shared resource.

File storage

File storage is built on top of block storage. It provides a higher-level abstraction to make it easier to handle files and directories. Data is stored as files under a hierarchical directory structure. File storage is the most common general-purpose storage solution. File storage could be made accessible by a large number of servers using common file-level network protocols like SMB/CIFS and NFS. The servers accessing file storage do not need to deal with the complexity of managing the blocks, formatting volume, etc. The simplicity of file storage makes it a great solution for sharing a large number of files and folders within an organization.

Object storage

Object storage is new. It makes a very deliberate tradeoff to sacrifice performance for high durability, vast scale, and low cost. It targets relatively “cold” data and is mainly used for archival and backup. Object storage stores all data as objects in a flat structure. There is no hierarchical directory structure. Data access is normally provided via a RESTful API. It is relatively slow compared to other storage types. Most public cloud service providers have an object storage offering, such as AWS S3, Google block storage, and Azure blob storage.

Domain Name System (DNS) lookup

DNS acts as an address book. It translates human-readable domain names (google.com) to machine-readable IP addresses (142.251.46.238).

To achieve better scalability, the DNS servers are organized in a hierarchical tree structure.

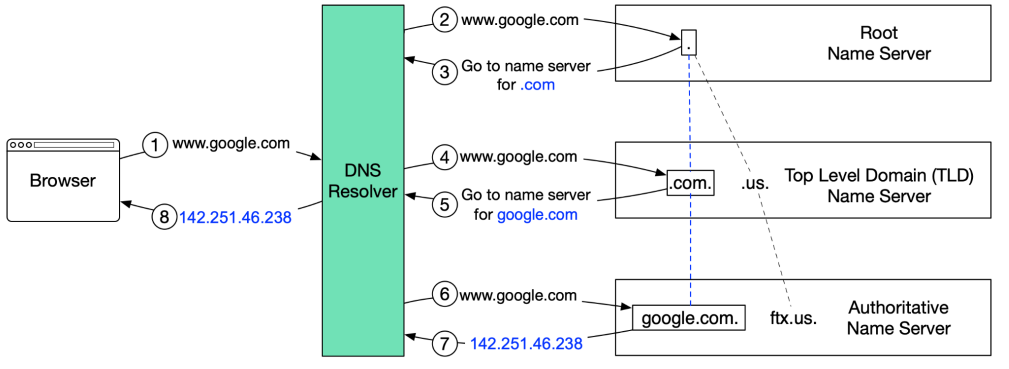
There are 3 basic levels of DNS servers:

1. Root name server (.). It stores the IP addresses of Top Level Domain (TLD) name servers. There are 13 logical root name servers globally.
2. TLD name server. It stores the IP addresses of authoritative name servers. There are several types of TLD names. For example, generic TLD (.com, .org), country code TLD (.us), test TLD (.test).

3. Authoritative name server. It provides actual answers to the DNS query. You can register authoritative name servers with domain name registrar such as GoDaddy, Namecheap, etc.

The diagram below illustrates how DNS lookup works under the hood:

How does DNS resolve IP



1. google.com is typed into the browser, and the browser sends the domain name to the DNS resolver.

2. The resolver queries a DNS root name server.

3. The root server responds to the resolver with the address of a TLD DNS server. In this case, it is .com.

4. The resolver then makes a request to the .com TLD.

5. The TLD server responds with the IP address of the domain's name server, google.com (authoritative name server).

6. The DNS resolver sends a query to the domain's nameserver.

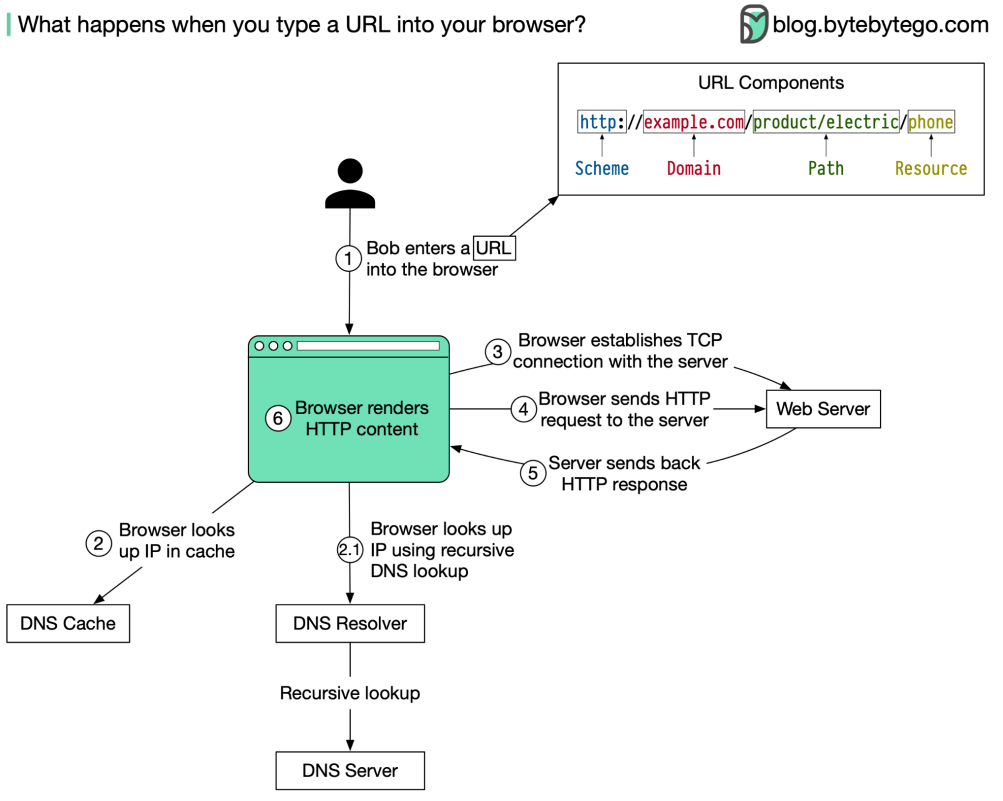
7. The IP address for google.com is then returned to the resolver from the nameserver.

8. The DNS resolver responds to the web browser with the IP address (142.251.46.238) of the domain requested initially.

DNS lookups on average take between 20-120 milliseconds to complete (according to YSlow).

What happens when you type a URL into your browser?

The diagram below illustrates the steps.



1. Bob enters a URL into the browser and hits Enter. In this example, the URL is composed of 4 parts:

- scheme - ***https***://. This tells the browser to send a connection to the server using HTTPS.
- domain - ***example.com***. This is the domain name of the site.
- path - ***product/electric***. It is the path on the server to the requested resource: phone.
- resource - ***phone***. It is the name of the resource Bob wants to visit.

2. The browser looks up the IP address for the domain with a domain name system (DNS) lookup. To make the lookup process fast, data is cached at different layers: browser cache, OS cache, local network cache and ISP cache.