Category

C/C++ Basics and Popular Library

1. OOP, encapsulation, inheritance, polymorphism, virtual function, pure virtual function, function overloading, function overriding, virtual function table (vtable), virtual destructor, function pointer, callback function, smart pointer — how can OOP be implemented in C? RVO, NRVO, std::move

The three core principles of object-oriented programming are **encapsulation, inheritance, and polymorphism**. **Encapsulation** involves bundling data and operations into objects and hiding internal details via access control while exposing necessary interfaces, enhancing safety and modularity. **Inheritance** allows subclasses to reuse properties and behaviors from parent classes, promoting code reuse and architectural extensibility. **Polymorphism** enables different objects to exhibit different behaviors through a common interface, often implemented via function overriding and dynamic dispatch, thus improving system flexibility and scalability. These three principles complement each other, forming a high-cohesion, low-coupling program structure that underpins modern software design.

Encapsulation is one of the fundamental principles of OOP. It refers to the practice of **bundling data members and the methods** that operate on these data within the same class, and **hiding implementation details through access control** (such as **private, protected, public**), **exposing only the necessary interfaces**. With encapsulation, the internal state of the class is not directly accessible from outside; it can only be indirectly manipulated through the methods provided by the class. This enhances the security, modularity, and maintainability of the program and is key to achieving "high cohesion, low coupling."

Inheritance allows a class (**subclass**) to **derive from another class (parent class) and automatically inherit its attributes and methods**. This promotes code **reuse** and **structural extension**. The subclass can directly use the parent class's members, or override its behaviors and add new functionality, thus modeling the process of "from general to specific." C++ supports both single and multiple inheritance, and the inheritance type (e.g., public, protected, private) determines the access rights to parent class members within the subclass. Inheritance forms the foundation for building class hierarchies, enabling polymorphism and decoupling modules.

Polymorphism refers to "**the same interface, different implementations**." It can be classified into compile-time polymorphism and runtime polymorphism. **Compile-time** polymorphism is achieved through function **overloading** and templates, while **runtime** polymorphism mainly relies on **virtual functions** and inheritance. Through a base class pointer or reference, a derived class’s overridden method is invoked dynamically at runtime, depending on the actual object type. Polymorphism enables programming to be interface-driven, enhancing code reuse and extensibility. It is a core concept for implementing the Open/Closed Principle and Dependency Inversion Principle.

A **virtual function** is a member function marked with the **virtual keyword**, designed to support **runtime polymorphism**. When a function in a base class is declared as virtual, a derived class can **override** it, and when called via a base class pointer or reference, the actual function invoked is determined by the type of the object at runtime. The virtual function mechanism is supported by the compiler through a virtual function table (**vtable**), which enables dynamic binding, the technical core of implementing polymorphic behavior.

A **pure virtual function** is a special type of virtual function whose declaration ends with **= 0**, indicating that the function has **no implementation** in the **base class** and must be **overridden by derived classes** to instantiate the class. A class that contains **at least one pure virtual function** is called an **abstract class**, and it cannot be instantiated directly; it serves as an interface. Pure virtual functions are commonly used to design a uniform interface specification, forcing subclasses to implement specific behaviors. They form the basis of interface-driven programming and design patterns such as Strategy.

Function overloading is a form of **compile-time polymorphism** where multiple functions with the same name but different parameter types or counts are defined within the same scope. The compiler automatically selects the matching version based on the arguments provided during the function call. Overloaded functions must differ in their parameter list; differing return types alone do not constitute overloading. Function overloading enhances the flexibility and readability of interfaces and is one of the key features of C++ that supports syntactic polymorphism.

Function overriding is the key feature of **runtime polymorphism**, where a subclass provides a new implementation for a **virtual function** inherited from a parent class. The function signature must match exactly. When the function is called via a base class pointer or reference, the version from the derived class is executed, enabling dynamic binding. C++11 introduced the override keyword to explicitly declare function overrides, which improves type checking and code readability, and is an important tool for interface-driven design.

A **virtual function table (vtable)** is an internal mechanism generated by the compiler to support **runtime polymorphism**. When a class defines a virtual function, the compiler creates a **vtable** that **stores pointers to the virtual functions**. Each object in memory contains a pointer to its class's vtable (vptr). When a virtual function is called through a base class pointer, the program uses the vptr to find the vtable and dynamically dispatch to the correct function. The vtable is the underlying mechanism for C++'s polymorphism, although it is invisible to the programmer, it controls the invocation path for polymorphic behavior.

A **virtual destructor** is a **destructor** declared as **virtual** in the base class to ensure that when a derived class object is deleted via a base class pointer, the derived class destructor is correctly called. In C++, if a class is designed to be "inherited and called polymorphically" (e.g., contains virtual functions), its destructor must be virtual. Otherwise, when using delete base_ptr to delete an object of the derived class type, only the base class destructor will be called, preventing the derived class resources from being properly released, leading to memory or resource leaks.

A **function pointer** is an abstraction provided by C/C++ to represent the address of a function, serving as a fundamental mechanism for implementing **callback mechanisms, strategy patterns, and decoupling communication between modules**. Essentially, a function pointer is a variable that **stores the entry address of a function**, allowing dynamic determination of which function logic to execute at runtime. The syntax for defining a function pointer requires specifying the function signature. For example, void (*fptr)(int) represents a function pointer that points to a function taking an int argument and returning void. Function pointers can be passed as parameters or used as elements in arrays or structures, making them a core mechanism for implementing strategy selection, state switching, and registry-based frameworks in early C projects. The fundamental goal of introducing function pointers is to abstract behaviors, allowing the caller to focus on the outcome of a function call without needing to know its specific implementation details, thus providing a foundation for decoupling between modules. In the absence of classes and virtual functions in C, function pointers are the only means to implement "interfaces" and "dynamic behavior switching." They are widely used in event dispatchers, state machines, plugin systems, and OS schedulers. A typical example is the POSIX thread library (pthread), where pthread_create() requires a function pointer to specify the entry function for a thread, and once the thread is created, it executes the function logic. This is a classic example of "passing control to the scheduled function," similar to callback registration. The disadvantage of function pointers is their poor type safety, lack of support for member function calls, and difficulty in carrying contextual data, which has led to the use of std::function in modern C++.

A **callback function** is an **application pattern for function pointers and represents a control inversion mechanism ("Inversion of Control"),** following the pattern of **"register -> store -> trigger."** The caller registers a function (or function object) with a lower-level module. When a specific event occurs, the lower-level module calls the function registered by the caller to complete the control reversal process. This is key for event-driven systems, asynchronous systems, and decoupling modules. The core advantage of the callback mechanism is that the caller only needs to define "how to handle something when it happens," without worrying about when or how the event will occur or be detected and scheduled; the lower-level framework decides the timing and triggers the callback. This allows for clear role separation and logical layering. In multi-threaded or event-driven systems, the **registered callback functions** are typically **not called immediately but are encapsulated** as task objects and added to a queue, where worker threads asynchronously schedule their execution at the appropriate time. This design is widely used in **network communication** (e.g., Redis, which registers read/write event callbacks using aeCreateFileEvent), thread pool task execution, message subscription systems, and GUI frameworks. In modern C++, callback functions are usually held using std::function in combination with lambda expressions or std::bind to bind parameters and context. To ensure thread safety and manage lifecycles, shared_ptr and weak_ptr are often used to implement callback validity checks and automatic deregistration. Through the callback mechanism, systems can achieve flexible plugin extension, asynchronous programming, and decoupling of callers and executors, making it an essential capability for understanding event-driven architectures and high-performance I/O models.

Private refers to the **most restrictive access control within a class**. Variables or functions marked as private can only be accessed by the member functions or friend functions of the class and cannot be directly accessed by external classes or objects. This mechanism is used to **encapsulate** implementation details and protect the internal state of the class from arbitrary modification. For example, the balance variable in a bank account class should be private so that users can only modify it through public interfaces (such as deposit or withdraw), preventing illegal direct modifications.

Private inheritance focuses on implementation reuse rather than semantic hierarchy, expressing an "is implemented in terms of" relationship. With private inheritance, **all public and protected members of the base class become private in the derived class, making them inaccessible to the outside.** This approach is often used when a class wants to reuse internal logic of another class without exposing its interface, similar to composition. For example, a stack class might privately inherit from deque to reuse its storage logic while hiding all other unrelated methods from the public API.

Public refers to members that are **exposed as interfaces of the class**. **Any object, function, or class can freely access public members.** Generally, constructors, getter/setter methods, and operation methods are made public to be called and reused. Public provides the capability for interaction between the class and external entities and is the basis for modularity and abstraction. A well-designed class should expose only the minimal necessary functionality via public interfaces, while other details are hidden in private or protected.

Public inheritance is the most commonly used inheritance type in C++, representing an "is-a" relationship between the base and derived classes. With public inheritance, **public**

members of the base class remain public in the derived class, and protected members stay protected, while private members remain inaccessible. This form of inheritance supports polymorphism, allowing derived objects to be manipulated via base class pointers or references. It is widely used to extend interface and behavior, for example: class Dog : public Animal means “a dog is an animal.” Features like virtual functions and virtual destructors rely on public inheritance to work correctly.

Protected refers to members whose **access rights lie between private and public**. **It cannot be accessed externally but can be accessed by derived classes (subclasses),** thus supporting inheritance access. This mechanism is particularly useful when the parent class provides default behavior and the subclass reuses and extends it without exposing the interface. For instance, an abstract base class’s template method might call a protected core processing function, and subclasses can override these functions to implement custom logic, but external users cannot access them.

Protected inheritance means that the derived class inherits the base class implementation but does not expose its interface publicly. In this mode, **both public and protected** members of the base class become **protected in the derived class,** while private members remain inaccessible. It does not support polymorphism, and base pointers cannot point to derived objects. Protected inheritance is typically used for internal code reuse without exposing the full API of the base class. For instance, class B may inherit class A’s utilities via protected inheritance but does not intend to be used as a type of A from outside.

A **smart pointer** is a **resource management** class template provided by the C++ Standard Library that follows the RAII (Resource Acquisition Is Initialization) principle. It automatically manages dynamically allocated memory to **prevent issues such as memory leaks**, **dangling pointers, and double deletions caused by manual delete operations**. A smart pointer is essentially a class object that overloads the * and -> operators to mimic the behavior of raw pointers. It automatically releases the managed resource in its destructor, enabling automatic and safe management of resource lifetimes.

std::unique_ptr<T>: **Exclusively owned**, **copy is forbidden and only move is allowed**. Suitable for scenarios where a resource has only one owner. unique_ptr is a lightweight smart pointer introduced in C++11 with no shared ownership overhead. Its core design is “unique ownership semantics”: a certain resource **can only be owned by one unique_ptr**. It **does not support copy construction or assignment** (i.e., unique_ptr<T> a = b; is invalid); it supports **move semantics**, and ownership can be **transferred** via std::move; it is suitable for resource management with strict lifetime constraints, such as **file handles, sockets, or database connections**.

std::shared_ptr<T>: **Shared** ownership, based on **reference counting mechanism**. Multiple pointers can share the same resource, and the resource is **destroyed when the last pointer is destructed**. shared_ptr is a smart pointer introduced in C++11 that **allows multiple shared_ptr instances to share the same heap resource**. Internally, it maintains a "control block" which contains: **use_count**: **how many** shared_ptr instances are sharing the resource; **weak_count**: the **number of weak_ptr referencing** the resource; **ptr**: **pointer to the actual managed resource**; **destructor pointer and deleter** (can be customized to define how the resource is released).

std::weak_ptr<T>: A weak reference pointer used with shared_ptr, **does not increase reference count**, used to solve cyclic reference issues. weak_ptr was introduced in C++11 to solve the cyclic reference problem caused by shared_ptr. It does not increase the reference count and does not own the resource, only observes whether the resource still exists. .lock(): attempts to obtain a valid shared_ptr from a weak_ptr (returns nullptr if the resource has already been released); .expired(): checks whether the resource has been destroyed.

Although C is not an object-oriented language, it is possible to simulate core OOP features such as encapsulation, inheritance, and polymorphism through **the use of structures, function pointers, and coding conventions**. **Encapsulation** can be achieved by **hiding the implementation of a structure** in a .c file and exposing only the operation functions, thus preventing direct access to internal fields. **Inheritance** can be implemented by **embedding one structure within another**, with the child structure having the parent structure as its first member, enabling upward typecasting. **Polymorphism** can be achieved by **defining function pointers within structures and building a virtual function table (vtable).** At runtime, when the function is called, it is dispatched to the corresponding implementation in the subclass via the base class pointer, achieving dynamic binding. This approach is widely used in low-level structures such as driver development, embedded systems, and protocol stacks.

RVO (Return Value Optimization) is a compiler-implemented performance optimization technique that eliminates the need to invoke copy or move constructors when returning a local object from a function. RVO allows the return object to be constructed directly in the caller's stack frame, avoiding the creation of intermediate objects.

NRVO (Named Return Value Optimization) is a subset of RVO. It applies in scenarios where the function returns a local object with a variable name.

std::move is a standard library function template introduced in C++11. It **casts a lvalue to an rvalue reference**, thereby enabling the invocation of a move constructor.

Semantically, it informs the compiler that the resources of the object can be “stolen” and do not need to be preserved.

2. How is C++ polymorphism implemented? What is the structure and calling process of the vtable? How can polymorphism be implemented in C?

C++ polymorphism is implemented via **virtual functions**. When a class contains **a virtual function, the compiler generates a vtable** (virtual function table), and each object stores a **pointer to this vtable (vptr).** When a virtual function is called, it is dynamically bound at runtime via the vptr->vtable.

C itself does not support object-oriented features, but **polymorphism** can be simulated using a **combination of function pointers and structures.** Specifically, a "base class structure" can be defined with function pointers acting as a "virtual function table." Different "derived class structures" inherit this base structure and implement their own version of the functions. At runtime, the function pointers in the structure are used to dynamically bind the appropriate behavior, thus achieving an effect similar to C++'s virtual functions. This approach is commonly used in driver development, operating system interfaces, and network protocol stacks, showcasing C's flexibility and low-level control capabilities.

3. Smart Pointers, the underlying implementation of smart pointers (unique_ptr / shared_ptr), and the reference counting mechanism?

Smart pointers are a resource management tool provided by C++ to **automatically manage the release of heap resources during an object's lifetime, preventing memory leaks and dangling pointers.** A smart pointer is essentially a class template that overloads the * and -> operators to behave like a regular pointer but automatically releases resources when the object is destroyed. **std::unique_ptr** represents exclusive ownership, meaning it **cannot be copied and can only be moved, making it suitable for scenarios** where an object has only **one owner**; **std::shared_ptr** represents shared ownership, with a **reference count** to track how many shared_ptr instances are sharing the same resource. **The resource is only destroyed when the last pointer is released.** **std::weak_ptr** is a weak reference used in conjunction with shared_ptr to avoid circular references, and it does not affect the reference count. The underlying implementation of shared_ptr involves allocating a control block that stores the reference count, weak reference count, and resource pointer. Every copy constructor or assignment operation atomically increases the reference count, and when the count reaches zero, the resource's destructor is called to free the memory.

4. wild pointers, dangling pointers, and null pointers?

A **wild pointer** refers to a pointer that is **uninitialized, inadvertently modified, or has gone out of bounds**. It typically **points to an unknown or invalid memory region** and poses a high risk of causing program crashes, undefined behavior, or severe security vulnerabilities. Common causes include **using local pointers without initialization**, buffer overflows that corrupt vtables or critical structures, returning the address of local variables, or accessing memory that has not been allocated or already freed.

A **dangling pointer** refers to a **pointer that was once valid**, but the **memory** it points to has already been **released** or is out of scope. Accessing it, such as dereferencing a pointer after free(p), results in undefined behavior.

A **null pointer** is explicitly set to **NULL or nullptr** to indicate that it currently points to no object. Accessing a null pointer causes a **segmentation fault** (segfault).

The fundamental difference among the three lies in their “pointing state”: wild pointers reference undefined or invalid memory, dangling pointers reference deallocated memory, and null pointers are intentionally unassigned.

To avoid pointer-related bugs, follow best practices: always **initialize pointers to nullptr at declaration**; **reset pointers to nullptr after calling delete or free**; never return addresses of local variables; avoid accessing members of temporary objects beyond their lifetime; use **smart pointers** (std::unique_ptr, std::shared_ptr) and the RAII paradigm to manage resources automatically; in C, use memset or explicit zeroing to manage lifecycle-sensitive structures; leverage tools like **AddressSanitizer** (ASan) or **Valgrind** to **detect illegal memory access** or use-after-free during development; and in multithreaded environments, never pass local addresses to threads or async callbacks where lifetimes might differ. Maintaining clear ownership semantics and access boundaries is essential for ensuring system reliability and memory safety.

5. Major differences between C and C++ in memory models, exception handling, and type systems?

C is procedural, lacks an exception mechanism, and has a loose type system. In contrast, C++ is object-oriented, supports exception handling and template-based generics, has a stricter type system, and its memory model includes object layout, constructor/destructor mechanisms, and polymorphism support.

6. What is the difference between deep copy and shallow copy? How would you implement a deep copy for a custom class?

The difference between deep copy and shallow copy lies in **whether heap-allocated memory resources are truly copied.**

A shallow copy means that the object's members are copied byte by byte, including the values of pointers (addresses), resulting in multiple objects sharing the same heap memory. If one object frees the resource, the other objects may become dangling pointers, causing crashes or memory errors. A deep copy, on the other hand, allocates new memory for the heap data and copies the contents, ensuring that each object has its own independent copy of the resources.

In C++, to implement deep copying for a custom class, you must explicitly define a copy constructor and assignment operator to ensure that pointer members are copied (not just the pointer addresses), while also correctly freeing any old resources to avoid memory leaks. This is crucial for achieving resource ownership and safe object copying.

7. What new features have you used in C++11/14/17/20?

C++11/14/17/20 introduced many new features that significantly improved syntax simplicity, type safety, performance optimization, and generic programming capabilities. Proper usage of these features can greatly enhance code quality and engineering efficiency.

C++11 is the cornerstone of modern C++, with notable features such as **auto type deduction, lambda expressions** for callbacks and function behavior encapsulation, **rvalue references and std::move** for resource move semantics, **smart pointers** (std::unique_ptr/std::shared_ptr) replacing raw pointers, **std::function** for unified callback encapsulation, **thread library** (std::thread/std::mutex) replacing POSIX threads for cross-platform concurrency, **nullptr** replacing NULL for type safety, **enum class** to avoid enum pollution, and **override/final** to clarify virtual function semantics.

C++14 added features like **generic lambda** (lambda auto parameters), make_unique for simplified smart pointer construction, and **binary literals**.

C++17 introduced **structured bindings** (auto [k, v]), if constexpr, std::optional, std::variant, std::any, and std::filesystem, which are widely used in engineering projects.

C++20 added concepts (template constraints), ranges for pipeline operations, and coroutines (co_await/co_yield), which significantly simplify asynchronous code flow. In real projects, I extensively use smart pointers for resource management, std::function combined with lambdas for event callback encapsulation, std::optional for non-mandatory return values, structured bindings for map iteration, rvalue references for network buffer management to improve memory reuse efficiency, and if constexpr for compile-time behavior switching in generic components. The usage of these features improves not just syntax but also code readability, safety, maintainability, and system resource efficiency.

8. Static variables inside and outside functions—do their addresses differ?

No. A static variable declared **outside a function** belongs to the **global static area**, while a static variable declared **inside** a function belongs to the **local static area**. Both have **different scopes** but share the **same lifetime**, meaning they persist for the entire program's execution. **However**, **their addresses do not match**, as they belong to **different data segments** allocated by the compiler.

9. What's the difference between memcpy and strcpy?

memcpy and strcpy are both used for **copying memory**, **but** they have completely **different purposes and behaviors.**

memcpy(void* dest, const void* src, size_t n) is a general-purpose memory copy function that copies n bytes from src to dest, and can be used for any type of data, including structures, arrays, and binary streams, without depending on content format.

On the other hand, strcpy(char* dest, const char* src) is specifically for copying **C-style strings**. It copies characters from src until it encounters the null-terminator (\0), making it suitable only for strings. The target buffer must be large enough to hold the entire string, including the terminator. If the source string lacks the null-terminator or the destination space is insufficient, strcpy can cause buffer overflow, while memcpy is better suited for copying fixed-length memory blocks and provides more precise control but can also be dangerous if length and pointers are not validated.

10. Describe malloc, free, new, and delete in detail.

malloc and free are memory management functions in C, while new and delete are operators in C++ that are fundamentally designed for dynamic memory allocation and deallocation.

malloc(size_t size) allocates raw memory of the specified size on the heap and returns a void* pointer, without constructing any objects, so it requires manual type casting.

free(void* ptr) releases previously allocated memory but invalidates the pointer, preventing its reuse.

In contrast, C++'s new **operator** not only allocates memory but also automatically calls the constructor to initialize an object, returning a pointer of the specific type. delete calls the destructor before releasing the memory. For arrays, C++ provides new[] and delete[] for creating and destroying objects in bulk. It is important to note that malloc/free and new/delete should not be mixed, as doing so can lead to memory leaks or program crashes. Internally, new/delete may also invoke operator new() and operator delete(), which can be customized by overloading. Generally, malloc/free is closer to low-level control, suitable for C or low-level resource management, while new/delete is more aligned with the object model and is the recommended approach in C++.

11. Static Binding and Dynamic Binding

Static binding (also known as early binding) is the mechanism where the function call address is determined at **compile time**. It applies to **non-virtual functions, function overloading, and template functions**. Since the function address is resolved ahead of time, it offers high execution efficiency and incurs no additional runtime overhead.

Dynamic binding (also known as late binding) is resolved at **runtime** through a **virtual function table (vtable),** and it is the core mechanism behind **runtime polymorphism** in C++. For dynamic binding to work, the function must be declared as virtual, and it must be invoked through a base class pointer or reference. When these conditions are met, the program uses the object's virtual pointer (vptr) to look up the actual function address in the vtable at runtime, enabling dynamic behavior dispatch. This is commonly used in interface-based programming and plugin architectures.

12. What’s the difference between pointers and arrays? Is an array name a pointer?

An array is a fixed-size, **contiguous** block of **memory**, and its name represents a constant pointer to its first element—it cannot be reassigned. In contrast, **a pointer** is a **variable** that holds a **memory address** and can be modified to point elsewhere. Although array names often "decay" to pointers in expressions, arrays and pointers are fundamentally different in nature: arrays are memory containers, while pointers are memory references. For example, int arr[5]; declares an array where arr is a constant address, but int* p = arr; declares a pointer that can later point to other locations. Arrays are typically stack-allocated with a compile-time fixed size, while pointers can be dynamically assigned to heap memory.

13. What’s the difference between an array of pointers and a pointer to an array?

An array of pointers means the **elements** in the array are **all pointers**, written as int* arr[3];, which is an array of three int pointers—commonly used to hold strings or struct pointers. A **pointer to an array** refers to a single **pointer pointing to an entire array**, written as int (*p)[3];, which points to an array of three integers. Their use cases are different: the former manages multiple individual objects, while the latter handles the whole array as one unit, especially useful when passing 2D arrays into functions. To distinguish, analyze the parenthesis: in *arr[3] you have an array of pointers; in (*p)[3] you have a pointer to an array.

14. FFmpeg

Keyword

FFmpeg is an open-source, cross-platform multimedia processing library that supports audio and video encoding/decoding, transcoding, muxing/demuxing, streaming protocol handling, filtering, and more. Its core components include libavcodec (codecs), libavformat (format demuxers/muxers), libavfilter (filter processing), and libswscale (image scaling). FFmpeg is widely used in video players, live streaming, recording and conversion, and media server systems. It supports both command-line tools and C/C++ APIs (FFmpeg API).

- **libavformat**: Handles multimedia container formats such as MP4, FLV, TS.
- **libavcodec**: Handles encoding and decoding (e.g., H.264, AAC).
- **libswscale**: Converts pixel formats and scales video frames.
- **AVFrame / AVPacket**: Core data structures representing media frames and encoded packets.

FFmpeg supports filtergraphs, timestamp synchronization, hardware acceleration (VAAPI, CUDA), and multithreaded processing.

1.14.1 How to decode a video using FFmpeg API?

Use the following sequence of function calls:
avformat_open_input → avformat_find_stream_info → find the video stream → use avcodec_find_decoder + avcodec_open2 to initialize the decoder → av_read_frame to read packets → avcodec_send_packet / avcodec_receive_frame to decode into AVFrame.

1.14.2 What's the difference between AVPacket and AVFrame?

AVPacket contains encoded/compressed data (e.g., H.264 NAL units), while AVFrame holds the decoded raw frame data (e.g., YUV images or PCM audio).

1.14.3 How to extract the audio stream from a video?

Open the input file with avformat_open_input, traverse AVStream to find the audio stream, then read and save the corresponding AVPacket data.

1.14.4 How are timestamps (PTS/DTS) used in FFmpeg?

PTS (Presentation Time Stamp) indicates when a frame should be displayed; DTS (Decoding Time Stamp) indicates decoding order. Synchronization requires converting timestamps with time_base.

1.14.5 How to stream with FFmpeg?

Use avformat_alloc_output_context2 to create an output context, configure AVStream encoder parameters, connect to RTMP/UDP via avio_open, and write packets with av_interleaved_write_frame.

1.14.6 How to handle videos with different pixel formats?

Use sws_getContext to create a conversion context, and sws_scale to convert from source format (e.g., YUV420p) to target format (e.g., RGB24).

1.14.7 How to crop or concatenate video using FFmpeg?

Use -ss and -t command-line options for cropping. For concatenation, use the concat protocol or demux/remux strategy.

1.14.8 How does FFmpeg support hardware-accelerated decoding?

Set up a hardware device context (e.g., AVHWDeviceContext), choose a hardware-supported decoder (e.g., h264_cuvid), and set it in AVCodecContext.

1.14.9 How to synchronize audio and video?

Compare the PTS values of audio and video frames, compute their difference, and synchronize playback using reordering or delay.

1.14.10 How are threads coordinated in FFmpeg encoding/decoding?

Typically, demuxing, decoding, and rendering are run in separate threads. Use thread-safe queues to transfer AVPacket and AVFrame for asynchronous, efficient processing.

15. Boost

Keyword

Boost is a collection of high-quality C++ libraries and is considered an essential extension of the C++ Standard Library. Boost covers algorithms, containers, threading, smart pointers, networking, serialization, regex, state machines, and more. Many Boost libraries have been included in the C++ standard (e.g., std::shared_ptr, std::function, std::regex).

- **boost::asio**: Asynchronous network programming (TCP/UDP, timers, coroutines).
- **boost::thread**: Cross-platform threading with locks, condition variables, atomics.
- **boost::filesystem**: Cross-platform filesystem operations.
- **boost::regex**: Regular expression processing.
- **boost::program_options**: Command-line and config file parsing.

1.15.1 Is boost::asio synchronous or asynchronous? How is async achieved?

It supports both. Asynchronous operations are implemented using io_context + handlers + async_xxx functions in an event-driven model.

1.15.2 How to implement a TCP Echo Server using boost::asio?

Create an acceptor to receive connections, use async_read / async_write for I/O, and register callbacks using lambdas or boost::bind.

1.15.3 What's the difference between boost::thread and std::thread?

boost::thread predates std::thread, has a similar API, but provides better thread group management and exception propagation. It is also more backward-compatible.

1.15.4 How to parse command-line arguments using boost::program_options?

Define options_description, parse using parse_command_line, and use store + notify to complete the parsing process.

1.15.5 What's the difference between boost::regex and std::regex?

boost::regex predates C++11 and supports more features such as Unicode and named groups. It is preferred in legacy systems.

1.15.6 What’s the difference between boost::bind and std::bind?

Functionally similar, but boost::bind supports more custom placeholders and is compatible with older compilers. Use std::bind for modern projects.

1.15.7 How does boost::shared_ptr differ from std::shared_ptr?

They are mostly similar, but Boost’s version integrates better with boost::enable_shared_from_this and includes weak_ptr for legacy compatibility.

1.15.8 How does boost::filesystem provide cross-platform path handling?

It offers a unified path class with functions like exists(), create_directory(), automatically adapting to platform-specific path formats.

1.15.9 How to implement timers in Boost?

Use boost::asio::steady_timer to set timeouts, and async_wait to handle timeout callbacks.

1.15.10 Which Boost libraries are useful for algorithm/data structure interviews?

Notable modules include boost::multi_index (multi-index containers), boost::graph (graph algorithms), boost::heap (heap structures), and boost::interval (interval management).

16. Qt

Keyword

Qt is a C++-based cross-platform GUI development framework that also supports non-GUI applications such as networking, threading, file I/O, database access, and XML parsing. Qt features a powerful signal-slot mechanism, an event system, and an integrated toolchain (Qt Designer, Qt Creator). It supports full-stack development from desktop apps to embedded systems.

Key Modules:

- **QtCore**: Core module (strings, containers, time, threads).
- **QtGui/QtWidgets**: UI and control widgets.
- **QtNetwork**: Networking APIs.
- **QtMultimedia**: Audio and video playback/capture.
- **QtConcurrent**: Concurrent programming.

- **QtQuick/QML**: Declarative UI language and rendering engine.

1.16.1 How does Qt's signal-slot mechanism work?

Based on Qt’s Meta-Object Compiler (MOC), signals and slots are connected at compile-time by generating additional C++ code. When a signal is emitted, Qt automatically calls the connected slot functions for decoupled event handling.

1.16.2 What is QObject used for, and why do many classes inherit from it?

QObject is the base class for Qt's object system, offering signal-slot support, event handling, object trees, and dynamic properties. Only classes inheriting QObject and using the Q_OBJECT macro can use these features.

1.16.3 What’s the difference between QThread and std::thread?

QThread is Qt’s thread abstraction, supporting signal-slot communication and event loops. std::thread is a basic C++11 thread without event mechanisms. QThread is preferred for Qt-based applications to integrate well with the UI and event systems.

1.16.4 How to safely update the UI from a background thread?

UI updates must occur in the main thread. Use signal-slot connections or QMetaObject::invokeMethod from the background thread to trigger UI changes in the main thread.

1.16.5 What are the use cases and principles behind QPainter?

QPainter is used for 2D rendering (lines, text, images). It wraps a device context and allows drawing on QWidget, QPixmap, or QImage. It's common for custom widgets and visualization.

1.16.6 How is internationalization (i18n) handled in Qt?

Mark translatable strings with tr() macro. Use Qt Linguist tools to generate .ts files, compile them into .qm files, and load with QTranslator at runtime for language switching.

1.16.7 How does the event delivery mechanism work in Qt?

Qt uses an event dispatch system (QEvent). Events (e.g., mouse, keyboard, timers) are placed in a queue and dispatched to the target object’s event() function, which then calls specific handlers like mousePressEvent().

1.16.8 What’s the difference between QML and QtWidgets?

QtWidgets is the traditional C++ UI framework, suitable for complex desktop apps. QML is a JavaScript-based declarative language ideal for modern, animated, and responsive UIs, often used in mobile and embedded systems.

1.16.9 How to implement a custom widget?

Inherit from QWidget or QFrame, override paintEvent() for custom drawing logic. You can add properties, layout support, and event handlers for complex behaviors.

1.16.10 What are Qt's memory management features?

QObject provides a parent-child hierarchy where deleting the parent automatically deletes all children, reducing manual memory management. Many Qt resources (e.g., QWidget) are also managed via smart pointer-like semantics to prevent memory leaks.

17. Memory Size of C++ Types and Structures (sizeof)

Type	sizeof (bytes)	Description
char	1	Fundamental type, always 1 byte by definition
bool	1	Boolean type, typically 1 byte
short	2	Short integer, 2 bytes
int	4	Integer, typically 4 bytes
long	8	Usually 8 bytes on Linux (64-bit)
long long	8	At least 8 bytes, for large integers
float	4	Single-precision floating point
double	8	Double-precision floating point
long double	16	Extended precision (implementation dependent)
void*	8	Generic pointer, 8 bytes in 64-bit systems
enum	4	Defaults to int type

union {int, char, double}	8	Size of largest member (double = 8 bytes)
struct {}	1	Empty struct occupies 1 byte as placeholder
struct {int x;}	4	Single int member
struct {int x; char y;}	8	With padding to align to 4 bytes
class A {int x; void f();}	4	Member function doesn’t affect size
class A {virtual void f();}	8	Adds virtual table pointer (vptr)
class A {}; // empty class	1	Empty class also occupies 1 byte
std::string	32	Depends on implementation, commonly 32 bytes

In C++, sizeof is a fundamental tool for determining how much memory a type occupies. Understanding the rules behind sizeof is crucial for memory optimization, data alignment, network protocol design, and structure serialization. From primitive types to class layouts, the size of a type is influenced by the following key factors:

1. **Primitive types have fixed sizes but are platform-dependent**: For example, int is typically 4 bytes on modern systems. On Windows, long is usually 4 bytes, while on Linux/Unix it is often 8 bytes. Pointer types (such as void*, int*, etc.) are 8 bytes on 64-bit platforms.

2. **Structure size is affected by member order and alignment**: The total size of a struct is not merely the sum of its members' sizes. The compiler may insert **padding bytes** between members to satisfy alignment requirements based on the most strictly aligned member. Poor ordering of members may lead to unnecessary memory waste.

3. **Classes and structs are fundamentally the same, but virtual functions introduce extra overhead**: When a class contains virtual functions, the compiler inserts a **virtual table pointer (vptr)** into each instance. This pointer typically

adds 8 bytes to the object size on a 64-bit system, thus increasing the result of sizeof(class).

4. **Empty types still occupy space**: Even an empty struct or class occupies at least **1 byte**. This is to ensure that each object instance has a unique memory address and does not overlap with others. The compiler inserts a single byte as a **placeholder padding byte**.
5. **STL container types are more complex internally**: For example, std::string is not just a raw character array. It includes pointers, size, and capacity fields. A typical implementation occupies **24 to 32 bytes**, not just the length of the actual string content. This makes it important to avoid misjudging its memory footprint.

Even if a structure contains no members, its sizeof is **not 0**, but **1 byte**. According to the C++ standard, every object must have a unique address, so the compiler allocates at least one byte as a placeholder, often referred to as a "padding byte".

For example: **struct A { int x; };**

This structure contains only one int member. On most modern platforms (such as 64-bit systems), int occupies 4 bytes. Since the structure's base address naturally satisfies the alignment requirement for int (typically 4-byte alignment), no additional padding is needed. Therefore, sizeof(A) == 4.

Another example: **struct A { int x; char y; };**

Here, int x occupies bytes [0–3], and char y occupies byte 4. To meet the overall alignment of the largest member type (int, 4-byte alignment), the compiler adds **3 bytes of tail padding** at the end. This makes the total size of the structure **8 bytes**.

Adding a **non-virtual member function** does **not** affect the result of sizeof(struct A). Member functions are part of the type definition, not part of the instance’s data layout. They are stored in the program’s code segment, not within the memory footprint of each object. Hence, adding a regular member function does not change the result of sizeof.

Padding refers to "invisible" bytes inserted by the compiler to ensure proper alignment for performance and correctness. Though these bytes do not store meaningful data, they are still present in the memory layout of the structure. They **can be read or written** by low-level operations such as memcpy, fread, or write. Therefore, in scenarios like network communication or binary file serialization, directly copying structure memory without considering padding may lead to inconsistent or incompatible data interpretation, especially when dealing with different platforms, endianness, or alignment rules.

Operating System & System Programming

A **Real-Time Operating System (RTOS)** emphasizes that "tasks must be completed within strict time constraints." The core focus is on **predictability** and **determinism**, making RTOS ideal for applications in embedded systems, automation, aerospace, and similar fields. RTOS provides high-precision timing, fast interrupt responses, and real-time task scheduling guarantees. Examples include FreeRTOS, VxWorks, and RTEMS.

In contrast, a **General-Purpose Operating System (GPOS)**, such as **Linux, Windows, and macOS**, focuses more on **throughput, resource utilization, and user experience**. The scheduling strategy tends to favor fairness and efficiency, but it lacks hard guarantees on response time. Therefore, RTOS is suited for systems that are extremely sensitive to delays, while GPOS is more appropriate for everyday desktop or server environments. The key distinction between the two lies in whether or not they can provide "time determinism."

The memory layout of a C program during runtime typically includes five major regions: **Text Segment**, **Data Segment**, **BSS Segment** (uninitialized data), **Heap**, and **Stack**. The **Text Segment** stores the program's machine instructions, is read-only, and shared. The **Data Segment** stores initialized global and static variables. The **BSS Segment** stores uninitialized global and static variables, and it is automatically zeroed out when the program starts. The **Heap** is used for dynamic memory allocation (such as malloc), and its space grows upwards as the program runs. The **Stack** is used for local variables and function call information, allocated by the operating system, and grows downwards. The entire memory layout grows from bottom to top with the heap and stack growing in opposite directions, and the program cooperates through these regions to manage runtime operations.

The **Heap** is the area of memory used for **dynamic memory allocation during program execution, managed by the operating system**. Developers allocate and deallocate memory using functions like malloc/free or new/delete. The heap is suitable for objects whose lifecycle is not fixed or whose size cannot be predicted, such as large arrays or buffers. Heap memory **does not automatically release** itself and must be explicitly freed by the programmer, which can lead to memory leaks or fragmentation. Its performance is generally lower than that of the stack.

The **Stack** is a **small block of memory automatically allocated by the operating system to store local variables, parameters, return addresses, and other information during function calls**. Memory on the stack is **automatically released** when functions exit, with variables being pushed and popped as functions are called and returned. Stack memory allocation is fast and convenient but limited in size, making stack overflow a potential issue.

A **global variable** is a variable defined **outside of functions** and stored in the static data segment. Its **lifetime** lasts for the **entire duration** of the program's execution. It is allocated when the program starts and released when it ends. Global variables can be accessed by multiple functions or files (via the extern keyword), but they increase module coupling, so they should be used cautiously.

A **static variable** is declared using the static keyword. Its **lifetime** is the same as the program's execution time, but its **scope** is **limited** to the function or file in which it is declared. A static variable defined within a function is initialized the first time the function is called and retains its value thereafter. A static variable defined outside a function has limited visibility and can only be accessed within that file, aiding in encapsulation. A static variable declared **outside** a function belongs to the **global** static area. A static variable declared **inside** a function belongs to the **local** static area. Both have the same lifetime but different scopes, and they are allocated in different locations within the data segment. Their addresses will not be the same.

Constants (or literal constants) are typically **stored in a constant area or read-only data segment (RO Segment)**, such as string literals or globally declared variables with the const keyword. Constants are determined at compile time and cannot be modified at runtime. Attempting to modify a constant results in undefined behavior or a segmentation fault. The presence of constants aids in program optimization and enhances code safety.

1. What is the difference between a process and a thread? Which is suitable for which scenario? Concurrency, parallelism, multithreading, and multiprocessing? How do different operating systems implement concurrency and parallelism?

A **process** is the **basic unit** of resource allocation in an operating system, with its own independent address space, memory, file descriptors, and other resources. A **thread** is the **smallest unit of program execution**. **Threads** within the same process **share** the **process's resources but have their own stack and registers**. **Processes** are **isolated** from each other, making them more **stable** and **secure**; threads, on the other hand, have lower context-switching overhead and more efficient communication, but a crash in one thread can affect the entire process. Processes are suitable for scenarios requiring high reliability and modular isolation, such as multi-tabbed browsers and the master-slave process architecture in databases. Threads are more suitable for scenarios with high performance requirements and frequent collaboration or resource sharing, such as high-concurrency web services, multithreaded downloads, and compute-intensive tasks.

Concurrency refers to a system's ability to handle multiple tasks within the same time period. Even if only one task can be executed at a time, the system can switch tasks quickly, making it appear as though multiple tasks are running simultaneously. This is commonly used in multi-task scheduling, I/O handling, event-driven systems, etc. For example, a single-core CPU can achieve concurrent execution of **multiple threads** through context switching. **Parallelism**, on the other hand, refers to **true simultaneity**, where multiple tasks are executed at the same time on multiple processors or cores. For example, a **multi-core CPU** can execute multiple threads simultaneously for large-scale computations. Concurrency focuses on the structure and scheduling of tasks, addressing the question of "how multiple tasks should be coordinated." Parallelism focuses on performance optimization, addressing the question of "how to complete tasks faster." In short, **concurrency is logical simultaneity**, and **parallelism is physical simultaneity**.

Multithreading and multiprocessing are two main ways of achieving concurrency or parallelism. **Multithreading** typically refers to multiple threads within the same process sharing memory space and concurrently executing tasks, which is a common method for achieving concurrency. When thread scheduling is reasonable and switching overhead is small, multithreading can also take advantage of multi-core CPUs to achieve parallel processing. **Multiprocessing** refers to running multiple independent processes, each with its own address space, commonly used in scenarios requiring process isolation and high security, such as web services and databases. On multi-core systems, multiple processes can also truly achieve parallel computation. In simple terms: concurrency is the goal, while multithreading/multiprocessing are the methods to achieve it; parallelism is a hardware capability, where multi-core CPUs enable true parallelism. In actual design, the choice of model should be based on resource isolation needs, communication overhead, and performance goals.

Operating systems support concurrency and parallelism through various methods. The most fundamental methods are thread and process scheduling, such as **pthread** in Linux, and thread APIs in Windows, which work with schedulers to achieve concurrency on single-core CPUs and parallelism on multi-core CPUs. To improve efficiency, operating systems also provide thread pools (such as GCD, ExecutorService, libuv), which reuse threads to reduce creation and destruction overhead. For I/O operations, non-blocking I/O and I/O multiplexing technologies (such as **epoll**, **kqueue**, **IOCP**) are used to avoid thread blocking and improve high-concurrency handling capabilities. In high-performance scenarios, **user-space coroutines** (such as libco, boost::asio, goroutines) are used to reduce context-switching overhead, or **task queues + work stealing** (such as Intel TBB, OpenMP, GCD) are used to fully utilize multi-core parallelism. In practice, most operating systems/languages/frameworks wrap these low-level mechanisms into user-friendly concurrency programming models to improve development efficiency and system performance.

2. Thread Communication and Process Communication Methods

In the operating system, different processes cannot share memory directly due to address space isolation, so they need to realize communication and collaboration through various **IPC** (Inter-Process Communication) mechanisms. Common process communication methods include: **pipe and named pipe (FIFO)**, which are based on kernel buffers and are suitable for small-scale, linear data transfer; **message queues** provide a structured, asynchronous messaging mechanism that is suitable for control-flow communication, but with system limitations; shared memory is the most efficient way of communicating, which allows multiple processes to map the same physical memory area, and has a fast read/write speed but must be used in conjunction with a mutual exclusion mechanism; **signals** are a type of signal that **allows multiple processes to share memory**, which is the most efficient way to communicate. must be used in conjunction with the mutual exclusion mechanism; signals are a kind of asynchronous event notification, suitable for process control or exception response; **sockets** not only support local process communication, but also the core of network communication, supporting **TCP/UDP** and cross-host transmission, which is the most common IPC method; **file mapping** (mmap) allows multiple processes to map the same file, realizing high-performance data sharing, which is the most common method in logging systems, logging systems, and other applications. File mapping (mmap) allows multiple processes to map the same file for high-performance data sharing, and is commonly used for logging systems, databases, and so on. Different IPC methods have their own trade-offs in terms of bandwidth, latency, scalability, security, and so on, and the selection should be based on the scenario.

Compared with processes, threads are lightweight scheduling units. Multiple threads share the address space of the same process, so they can directly access global variables and heap memory, but they also have data consistency problems caused by concurrent access. **Synchronization** between threads usually use locking mechanisms, including **mutex locks** (mutex) used to protect the critical area, to ensure that only one thread access to shared resources at the same time; **read/write locks (rwlock)** allows multiple threads to **read concurrently, write operations are mutually exclusive**, suitable for reading more and writing fewer scenarios; **conditional variables** are used to **wait** for the conditions of the **inter-thread wake-up**, often used in conjunction with the mutex locks, to achieve the coordination of the logic of the inter-threads; **Signal volume** (**semaphore**) is a generalized synchronization primitive, can be used to limit the number of access to resources or to achieve inter-thread mutual exclusion, support for cross-process use; In addition, **inter-thread communication methods also include atomic variables, spin locks, barriers and other means.** To improve development efficiency and security, modern C++ provides RAII-style lock managers (e.g., std::lock_guard), thread libraries, and intelligent concurrency primitives to avoid manually managing lock releases. Meanwhile, when multi-threaded access to global variables, shared memory and other resources, be sure to synchronize and protect them, otherwise you will face data contention, deadlock, performance jitter and other concurrency problems.

3. Thread communication vs. process communication?

Due to address space isolation between processes, they cannot directly share memory. Therefore, communication must rely on intermediary mechanisms provided by the operating system, such as kernel buffers, shared memory mappings, system calls, etc. The core goal of IPC (Inter-Process Communication) is to achieve data exchange and event synchronization while preserving isolation. Different methods vary in terms of structural complexity, performance, and application scenarios, and the appropriate choice should be made based on application requirements.

(1). **Pipe / FIFO** — Suitable for passing linear data streams between parent and child processes. Anonymous pipes are ideal for transferring simple data between parent and child processes, such as when a child process reads the output from the parent process after a fork. In practice, shell command pipes (e.g., **ps aux | grep nginx**) are implemented using anonymous pipes to connect standard input and output between multiple processes. **FIFO** (named pipes) can facilitate communication between unrelated processes, such as two independent processes exchanging instructions or status information through a pre-agreed path (e.g., /tmp/myfifo). It is suitable for simple IPC scenarios with clear requirements.

(2). **Message Queue** — Suitable for structured, event-driven asynchronous communication. Message queues are used for asynchronous, structured information transfer between processes and can classify messages by type, making them ideal for "control message bus" scenarios. For example, a daemon process may use a message queue to receive "task completion notifications" or "exception status reports" from child processes, or a scheduler may manage job submissions via a message queue. Its non-blocking, asynchronous features are ideal for event-driven systems and scheduling systems, though it is limited by kernel message size and is not suitable for large data transfers.

(3). **Shared Memory** — Suitable for high-frequency large data sharing and state sharing. Shared memory, with its zero-copy nature, is ideal for high-speed data exchanges, such as video frame data, image buffers, database page caches, or shared log buffers. For example, two image processing processes, one for camera acquisition and the other for image processing, can exchange frame data directly via shared memory. Another example is a log aggregation module in a log collection system, which aggregates logs from multiple processes into a shared buffer, avoiding frequent disk writes. It is important to use locks to ensure access consistency.

(4). **Signal** — Suitable for asynchronous event notification and process control. The signal mechanism is useful for asynchronous event notifications between processes or between the system and processes. For example, a parent process may send SIGTERM to a child process for graceful exit, or a child process may send SIGUSR1 to notify the parent process that "tasks are completed." In servers, SIGCHLD can notify the parent process to collect child process resources. Additionally, timeout handling is often implemented using alarm() + SIGALRM. While signals are not suitable for large data exchanges, they are indispensable for process lifecycle control and error interruption handling.

(5). **Socket (including UNIX Socket)** — Suitable for inter-host/module communication and network service development. Sockets are the core method for building distributed systems and multi-process module communication. They support both local communication (UNIX Domain Socket) and network communication (TCP/UDP). They are suitable for client-server architectures. In multi-language systems (e.g., a Python-based collector and a C++-based analyzer), sockets can also act as a universal bridge for module decoupling. Additionally, UNIX local sockets support file descriptor passing, making them suitable for "hot-swappable" communication needs.

(6). **mmap (Memory-mapped files)** — Suitable for caching, database, and configuration sharing, and persistent sharing. mmap is ideal for sharing file-based data or high-frequency read-only data in caching scenarios. For example, multiple log processing processes can use mmap to map a temporary cache file for data exchange, or a configuration center can use memory mapping to share configuration files with all child modules. Compared to shared memory, mmap has advantages in persistence, cross-restart recovery, and file rollback, especially in embedded systems or database middle-layer scenarios.

Since threads are within the same process and share memory space, communication does not require intermediary mechanisms. Instead, it is achieved through shared variables and synchronization controls. The key is ensuring data consistency and thread safety during concurrent access. Below are common thread communication methods and their typical application scenarios.

(1). **Mutex** — The most basic method for protecting shared resources. A mutex is a synchronization primitive that ensures exclusive access to a critical section, **allowing only one thread at a time to hold the lock and access shared data**. For example, when multiple threads access a global map, modify a counter, or operate on a shared connection pool, the resource reads and writes must be protected by std::mutex (or pthread_mutex) to prevent data races. C++11 provides std::lock_guard and std::unique_lock to implement the RAII lock release mechanism, improving code robustness. Mutex is a core primitive in thread-safe programming.

(2). **Condition Variable** — Suitable for **waiting and notification mechanisms**, such as the **producer-consumer model**. Condition variables are used for waiting/waking mechanisms between threads. When a thread needs to wait for a condition (e.g., a queue being non-empty), it will block on a condition variable until another thread triggers a wake-up using notify_one() or notify_all(). A typical application is the producer-consumer model: the producer thread wakes up the consumer thread after putting in a task; the consumer waits for the task queue to be non-empty before proceeding. C++ provides std::condition_variable, which, when combined with mutex, enables efficient thread coordination. It is suitable for passive waiting logic when data arrives or resources become available.

(3). **Read-Write Lock (RWLock)** — Suitable for **high-concurrency read-heavy scenarios**. A read-write lock is a synchronization mechanism that **allows multiple threads to read concurrently but ensures mutual exclusion for write operations**. It is suitable for scenarios where reading far outweighs writing, such as caching systems, configuration tables, and data statistics modules. In C++, this can be achieved using shared_mutex/shared_lock (C++17) or pthread_rwlock_t. Multiple threads can read data concurrently, and only write threads block other threads. Compared to regular mutexes, it reduces the waiting delay for read operations and is a common tool in performance optimization.

(4). **Semaphore** — Used for resource access **rate limiting or thread synchronization counting**. A semaphore is a counting synchronization primitive that controls the number of threads accessing a resource concurrently. For example, in connection pools, task slots, or memory block management, a fixed-size semaphore is used to control the maximum number of concurrent threads accessing a resource. C++20 provides std::counting_semaphore, and in POSIX, APIs like sem_init, sem_wait, and sem_post are used to manage semaphores. Semaphores can also be used for one-time notifications between threads, such as notifying the main thread to continue execution after the initialization of a child thread is complete.

(5). **Spinlock** — Suitable for small lock granularity and very short critical section execution time. A spinlock is a **non-blocking lock** where a thread does not suspend but spins on the CPU when it cannot acquire the lock. It is suitable for high-frequency scenarios where the lock hold time is very short, such as writing to a log buffer, updating a status bit, or atomic counting. Spinlocks avoid thread-switching overhead but occupy CPU resources. In C++, a simple spinlock can be implemented using std::atomic_flag. It should be used with consideration of the critical section’s complexity, as improper usage could lead to performance degradation.

(6). **Atomic Operations** — Used for **lock-free concurrent operations**, suitable for **lightweight synchronization scenarios**. Atomic operations are unit operations provided at the CPU level that cannot be interrupted. They are commonly used for lock-free concurrency control, such as atomic counting, incrementing, decrementing, and CAS (Compare-and-Swap). C++11 provides std::atomic<T> for atomic read/write operations and the compare_exchange_xxx series of functions, enabling the creation of lock-free stacks, queues, and other high-performance components. Atomic operations are fast, without the risk of deadlocks, and do not rely on mutexes, but they are best suited for simple logic and lightweight data structures.

(7). **Barrier** — Used for **phase synchronization between threads** (e.g., aligning parallel processing). A barrier is a phase synchronization mechanism that requires all threads to "meet" at a specific point, only continuing execution once all threads have arrived. It is suitable for parallel data processing, image block processing, and phase aggregation scenarios. C++20 introduces std::barrier, and before that, it was commonly implemented manually using counting + condition variables or semaphores. It is suitable for fixed thread counts and tasks executed in stages, such as multi-threaded MapReduce.

(8). **Thread-Safe Queue** (with or without locks) — Suitable for task dispatching and asynchronous message handling. A thread-safe queue is a high-level communication component that typically uses mutex or lock-free methods (such as circular queues + atomic operations) at the lower level. It is commonly found in task queues, work queues, and thread pool task submissions. Producer threads place tasks using push(), while consumer threads retrieve tasks using pop(). Combined with condition variables, it can implement asynchronous blocking or timed wake-up. In modern server architectures, thread-safe queues are foundational for asynchronous processing and high-concurrency decoupling.

4. What is socket programming? How do you implement a high-concurrency server?

Socket programming is a communication programming model based on **network sockets, allowing processes on different hosts or on the same host to exchange data via TCP or UDP protocols**. It encapsulates the details of underlying network protocols and provides a set of interfaces for creating sockets, binding addresses, listening, accepting connections, and sending/receiving data. Common models include TCP (connection-oriented, reliable transmission) and UDP (connectionless, suitable for real-time scenarios). Socket programming is widely used in client-server architectures, such as web services, chat applications, and remote control.

To implement a high-concurrency server program, the key is efficiently managing a large number of connections and I/O operations. Typically, I/O multiplexing techniques (such as **select, poll, epoll**) are used to avoid blocking calls and handle thousands of connections in a single thread or a small number of threads. Thread pools or coroutines can also be used to increase throughput, and techniques like non-blocking sockets and connection reuse (e.g., SO_REUSEADDR) further optimize performance. On Linux, epoll + non-blocking I/O is a mainstream combination, and when paired with a well-designed event-driven architecture, it can achieve high-performance and high-concurrency network servers.

5. What is `select`, `poll`, `epoll`? Have you used `epoll`?

select, poll, and epoll are I/O multiplexing mechanisms provided by Linux for monitoring multiple file descriptors (such as sockets) to check if they are readable, writable, or have errors, enabling a single thread to handle multiple connections. **select** was the first to be introduced but has a **limited number of file descriptors (by default, 1024).** Every time it is called, the FD set must be copied from user space to kernel space, which reduces efficiency. **poll removes this limitation**, but it still requires **linear traversal** of all file descriptors each time. **epoll** is a Linux-specific, **highly efficient** mechanism that supports "**event-driven**" programming (only notifying active connections). File descriptors are registered once and do not need to be passed repeatedly, with efficiency approaching **O(1)** as the number of connections increases. It is especially suitable for high-concurrency scenarios.

The **C10K problem** refers to the challenge of efficiently handling ten thousand concurrent connections on a single server. It represents the performance bottleneck of early network servers. The traditional one-thread-per-connection model suffers from high memory usage and context-switching overhead in high-concurrency situations. epoll, with its event-driven model and non-blocking I/O, provides a solution, allowing a single thread to handle thousands of connections. When studying C10K, I focused on using epoll, including functions like epoll_create, epoll_ctl for event registration, and epoll_wait for waiting for events to trigger, further enhancing performance by using non-blocking sockets and edge-triggered mode. I am familiar with using epoll to build high-concurrency server models.

6. What is memory leak? How do you debug it? What tools help with detection?

A **memory leak** occurs when a program **allocates heap memory but fails to release** it in a timely manner, losing all references to that memory, which means the memory cannot be used or reclaimed. Although a memory leak may not immediately cause a program crash, it can cause memory usage to steadily increase, potentially leading to system performance degradation, memory exhaustion (**OOM**), or even process crashes over time. Memory leaks are a significant concern in server-side and embedded system development.

The key to debugging memory leaks is identifying "where memory is allocated and why it was not released." During development and testing, tools like **Valgrind** are used for precise offline detection, providing detailed call stack information and leak reports. **AddressSanitizer** uses compile-time instrumentation to perform real-time checks on memory operations, with relatively low performance overhead, making it suitable for integration into CI processes for frequent detection. For complex projects, developers can wrap memory operations like malloc/free to log call stacks and context information, tracking unreleased objects through logs or hash tables to help pinpoint the exact location of leaks.

For deeper analysis, tools like **Massif** can generate memory usage graphs over time, helping identify which modules or function blocks cause memory spikes, thus optimizing memory structure and release strategies. For Windows platforms, tools like **Visual Leak Detector** can assist with visualizing leaks during development.

In **production environments**, where high-overhead debuggers cannot be used, heap profiling tools like **jemalloc** or **tcmalloc** are often used to **capture heap snapshots and analyze memory growth trends through diffs**. Monitoring memory-related metrics (e.g., RSS, heap size, object count) can help detect potential leaks and track memory allocation sources with instrumentation on key resource paths. Additionally, system commands like **/proc/[pid]/smaps, pmap, and top can be used to check real-time memory distribution**, while when **an OOM event occurs**, **dmesg -T** can be used to **confirm the time of memory exhaustion and which process was killed by the OOM Killer**, helping trace the leak's aftermath and critical paths. While dmesg -T is not directly used to locate the leak source, it is an important auxiliary tool for diagnosing system-level anomalies caused by memory leaks in production environments.

7. What is memory fragmentation? Why does it happen? How to avoid it?

Memory fragmentation refers to the situation where **frequent memory allocation and deallocation cause many non-contiguous, unused memory blocks** scattered across the heap, making it difficult to satisfy requests for larger contiguous memory allocations, even though the total available memory space is sufficient. Memory fragmentation is divided into **external fragmentation** (free memory that is not contiguous) and **internal fragmentation** (allocated memory blocks that are larger than needed).

The main cause of fragmentation is the interleaving of memory allocation requests of different sizes with irregular deallocation orders, combined with the inability of the memory allocator to compact the memory. Over time, this leads to allocation failures or performance degradation due to severe fragmentation. In high-reliability systems, controlling fragmentation is an important part of memory management optimization.

To reduce or avoid memory fragmentation, both memory allocation strategies and program design approaches can be applied. First, using **memory pools** or **object pools** is a common method. This involves pre-allocating a large contiguous block of memory and dividing it into fixed-sized blocks for reuse, thus avoiding frequent allocation and deallocation. Secondly, frequent allocation and deallocation of objects of varying sizes should be avoided. Strategies like object reuse, delayed deallocation, and unified lifecycle management can reduce memory fluctuations. Additionally, using specialized memory allocators (e.g., **jemalloc**, **tcmalloc**) can improve allocation efficiency and automatically perform fragmentation cleanup. In embedded or long-running systems, periodically restarting submodules, migrating data, and proactively reclaiming memory can maintain a compact memory structure and prevent uncontrolled fragmentation accumulation.

8. Deadlock, Race Conditions, Zombie processes, Orphan Process

Deadlock refers to a state where two or more threads or processes cannot proceed because they are each waiting for the other to release resources. This situation satisfies four necessary conditions: **mutual exclusion, hold and wait, no preemption, and circular wait**.

Race Condition occurs when multiple threads or processes access shared resources concurrently without proper synchronization, leading to unpredictable outcomes or even program crashes due to the lack of certainty in execution order. It is typically solved through locking mechanisms or atomic operations.

A **Zombie Process** refers to a **child process** that has **finished** executing and **exited**, **but** its exit status information remains in the system, **waiting for the parent process** to collect it via wait() or waitpid(). If the parent process does not process this in time, many zombie processes may remain in the system, occupying process table slots, and potentially preventing new processes from being created.

Orphan Process: An orphan process refers to a **child process** that is **still running** after its **parent process** has **already exited**. In this case, the child process is adopted by the operating system (usually by init or systemd), which takes responsibility for cleaning up its resources once it finishes. Orphan processes do not pose a threat, as the system handles them properly and automatically.

9. Linux boot process and how to debug unfamiliar kernel panics?

From the moment the system is powered on or rebooted, the BIOS/UEFI initializes the hardware, **loads** the **bootloader** (e.g., **GRUB**), and then loads the kernel image (vmlinuz) into memory to start. After the kernel starts, it initializes **memory management, CPU cores, loads drivers, mounts the root filesystem, and eventually executes the first user-space process /sbin/init (or systemd), which continues to launch system services and user processes.** The entire process can be broken down into three phases: the bootloader phase, the kernel startup phase, and the user space initialization phase, which is the complete path for the system to go from bare metal to a usable state.

To debug a kernel panic, the first step is to collect panic information, including the call trace on the screen, the error module, and the line of code where the panic occurred. If the panic is reproducible, enable debug symbols during kernel compilation (CONFIG_DEBUG_INFO=y), and use dmesg, /proc/kmsg, or serial console logs for capture. You can use addr2line to analyze the symbol addresses and pinpoint the specific function and code line. If the panic is caused by a driver or third-party module, insert printk or dynamic debugging in the module. Additionally, crash dump tools (such as **kdump** or **crash**) can be used for crash analysis. The troubleshooting process typically involves analyzing memory access errors, null pointer dereferencing, out-of-bounds operations, interrupt deadlocks, lock contention, and illegal instructions, which requires patience and experience.

Keywords and Concepts:

Deadlock: threads/processes block waiting on each other's resources.

Livelock: not truly blocked but continuously retrying without progress.

Starvation: thread gets no CPU time due to unfair scheduling.

Race condition: unordered concurrent access to shared data causes bugs.

Critical section: shared resource access code requiring protection.

Atomic operation: uninterruptible unit of execution for concurrency.

Semaphore: sync primitive for controlling concurrent threads.

Mutex: exclusive lock for critical sections.

RWLock: concurrent read/mutual exclusive write—suits read-heavy workloads.

Condition variable: wait/notify coordination between threads.

Thread starvation and priority inversion: high-priority threads waiting for resources held by lower-priority threads.

Process: execution unit with isolated address space.

Thread: smallest execution unit within a process, shares memory.

User mode vs. kernel mode: CPU privilege levels, switching required for system calls.

Interrupt: hardware-triggered event that interrupts CPU execution for handling.

System call: interface where user programs request services from the kernel.

Zombie process: exited child process not yet collected by parent.

Orphan process: child process whose parent exited, adopted by init.

Context switch: CPU switching between threads/processes, involves saving/restoring register states.

Thread pool: reuse pre-created threads to reduce creation/destruction overhead.

I/O multiplexing (epoll, select, poll): single thread handles many I/O sources concurrently.

Memory leak: unreleased and unreachable heap memory, wastes resources.

Segmentation fault: program crash from illegal memory access.

Computer Networks

1. Describe the OSI 7-layer or TCP/IP 5-layer model:

The **OSI seven-layer** model, from bottom to top, consists of: **Physical Layer, Data Link Layer, Network Layer, Transport Layer, Session Layer, Presentation Layer, and Application Layer**. It is a theoretical model used to standardize communication between different network devices. In practice, the more commonly used model is the **TCP/IP five-layer model**, which corresponds closely to the OSI model but with some simplifications. The five layers are: **Physical Layer, Data Link Layer, Network Layer, Transport Layer, and Application Layer**. Each layer is responsible for specific functions, such as the Network Layer being responsible for routing and forwarding, the Transport Layer for end-to-end communication, and the Application Layer for user interaction. This model helps us understand the division of labor in network protocols and the data transmission path.

The OSI seven-layer model, from bottom to top, consists of the Physical Layer, Data Link Layer, Network Layer, Transport Layer, Session Layer, Presentation Layer, and Application Layer. The **Physical Layer** is responsible for bit transmission, such as **network cables, optical fibers, and electrical signals**. The **Data Link Layer** handles **frame transmission and error detection**, with common protocols like Ethernet and ARP, and **corresponding devices** like **switches**. The **Network Layer** is responsible for **routing and addressing**, with typical protocols like **IP and ICMP**, and corresponding devices like **routers**. The **Transport Layer** manages end-to-end communication control, with protocols like **TCP and UDP**, ensuring data reliability and order. The **Session Layer** handles session establishment, management, and termination, such as **remote login**. The **Presentation Layer** is responsible for **data format conversion and encryption/decryption**, such as **JPEG, TLS, and ASCII encoding**. The **Application Layer** directly **interacts with users**, providing application services such as **HTTP, FTP, and DNS**. This model helps us understand the complex communication process of network systems in a layered manner.

TCP is a connection-oriented, **reliable transmission protocol** that establishes a connection through a **three-way handshake**, ensuring data is **transmitted in order and without error**. It is commonly used in applications where integrity is crucial, such as **HTTP**, FTP, etc. **UDP**, on the other hand, is a **connectionless protocol that does not guarantee order or reliability**, making it suitable for real-time applications that require **lower latency**, such as **voice calls, video streaming**, DNS, etc. HTTP is a stateless protocol based on TCP used for request-response communication between browsers and servers, like transmitting HTML pages, images, and videos. **HTTPS** is an enhanced version of **HTTP that adds a TLS/SSL encryption layer**, providing identity authentication and data encryption to secure communication. It is widely used in sensitive information transmission, such as payment and login. **RTSP** (Real-Time Streaming Protocol) is an **application layer protocol** designed specifically for **audio and video streaming**. **RTSP 1.0** uses text-based communication and establishes control connections over TCP, but data is usually transmitted via RTP. The process is relatively complex, and recovery from interruptions is not straightforward. **RTSP 2.0** is a binary protocol, modernized with improvements in state management, support for pipelining, reduced handshake times, and better real-time performance and resource efficiency, making it ideal for the next-generation streaming systems.

2. Describe TCP 3-way handshake and 4-way teardown. What's the role of TIME_WAIT?

The three-way handshake is the process of **establishing a connection**. It begins with the client sending a **SYN**, the server replying with **SYN+ACK**, and then the client sending **ACK**. Once the connection is established, both sides can communicate.

The four-way handshake is used to **disconnect the connection**. The side initiating the close sends a **FIN**, the receiving side responds with **ACK**, the receiving side then sends **FIN**, and the initiating side replies with **ACK** to complete the disconnection.

TIME_WAIT is the waiting state entered by the side that initiates the connection closure. It lasts for about 2 minutes to ensure that the final ACK from the other side is reliably received and to avoid delayed data from old connections interfering with new ones.

3. What are IPv4, IPv6, HTTP, HTTPS, TLS, NAT, DNS, IP Datagram, and VPN?

IPv4 is the fourth version of the Internet Protocol and is the most widely used network layer protocol on the internet today. It uses a **32-bit address** format, typically represented in **dotted** decimal notation, such as 192.168.1.1. IPv4 provides approximately **4.2 billion** unique addresses, but due to the scarcity of public addresses, NAT (Network Address Translation) is widely used to alleviate address shortages. The IPv4 header structure is relatively simple and is widely used in traditional networks, home routers, and most enterprise networks, but as the global Internet of Things and mobile devices grow rapidly, the address space is gradually becoming insufficient.

IPv6 is the **successor to IPv4** and uses a **128-bit address** design, theoretically offering 2^{128} unique addresses, solving the IPv4 address exhaustion problem. IPv6 addresses are represented in **hexadecimal**, separated by **colons**, such as 2001:0db8:85a3::8a2e:0370:7334, and support abbreviation rules. Compared to IPv4, IPv6 natively supports the IPSec security protocol, removes the concept of broadcast, introduces multicast and anycast, and replaces ARP with the Neighbor Discovery Protocol (ND). IPv6 was designed to build a scalable, highly secure internet without the need for NAT, but due to compatibility issues and deployment costs, it is currently promoted through a dual-stack strategy and transition mechanisms.

HTTP (Hypertext Transfer Protocol) is the foundational communication protocol for the web, used for data transmission between clients (browsers) and servers, such as HTML pages, images, and videos. It is a stateless protocol that transmits data in plaintext and typically uses **port 80**. Due to its plaintext nature, HTTP cannot prevent man-in-the-middle attacks, data tampering, or identity spoofing, making it less suitable for security-sensitive scenarios. However, HTTP is simple, efficient, and well-suited for low-latency applications where no sensitive information is involved, such as public news websites and static resource loading.

The **HTTP protocol consists** mainly of **request messages** and **response messages**, which include methods (e.g., **GET, POST, PUT, DELETE**), status codes (e.g., **200 OK, 404 Not Found**), request and response headers (e.g., Host, User-Agent, Content-Type, Cache-Control), and body content. HTTP supports mechanisms such as **content negotiation**, **chunked transfer encoding**, **keep-alive connections**, and **cache control**. Overall, it is a stateless, plaintext transmission protocol based on TCP communication, but it is gradually transitioning to HTTP/2 and HTTP/3 to support multiplexing, header compression, flow control, and other optimizations.

HTTPS is the secure version of **HTTP**, incorporating a **TLS/SSL encryption layer**, and typically uses **port 443**. HTTPS performs a TLS handshake before communication, negotiating encryption algorithms and completing key exchanges, after which all HTTP data is transmitted over an encrypted channel. HTTPS effectively prevents man-in-the-middle attacks, eavesdropping, and data forgery, and is widely used for sensitive information transmission, such as in login, payment, banking, and e-commerce scenarios. While HTTPS adds overhead due to encryption and handshake time, modern hardware can generally make this negligible, making it the standard for secure internet communication.

HTTPS, built on top of HTTP, adds a TLS security protocol layer. Its communication content remains standard HTTP messages, but these are encrypted via TLS before transmission. **The TLS layer** includes mechanisms such as **asymmetric key exchange (RSA or ECDHE)**, **symmetric encryption (AES or ChaCha20)**, message authentication codes (MACs), digital certificates (X.509), and handshake protocols (TLS Handshake). During the client-server connection establishment, a TLS handshake occurs to negotiate algorithms and keys, verify the server’s identity (and optionally the client’s), and then encrypted HTTP data transmission begins. Thus, HTTPS includes encryption algorithms, certificate verification, TLS session caching or recovery, and other security-related modules in addition to the standard HTTP content.

The **TLS handshake** is the process for establishing a secure HTTPS connection, used to negotiate encryption algorithms, exchange keys, and verify identities. For example, with **TLS 1.2**, the process includes: the client sending a ClientHello with supported encryption algorithms, random numbers, etc.; the server responding with ServerHello, a digital certificate, public key, random numbers, etc.; the client verifying the certificate's validity and generating a "pre-master key," which is then encrypted using the server’s public key; the server decrypts the pre-master key using its private key, and both the client and server independently compute a symmetric encryption key based on the random numbers and pre-master key; finally, both sides verify the success of the handshake with a Finished message and enter the encrypted communication phase. **TLS 1.3** further simplifies the handshake process, **reduces RTT**, and **enhances security and efficiency**.

NAT (Network Address Translation) is a technique for translating network addresses, primarily used to **map private internal IP addresses (such as 192.168.x.x) to public IP addresses**, enabling multiple internal devices to share a single public IP to access the internet. NAT is commonly used in **home and business routers**. It not only conserves IPv4 address resources but also provides a degree of security isolation. The basic principle of NAT is that when an internal host sends a data packet, the NAT gateway replaces the source IP address with the public address and records the mapping relationship between the original IP and port in a NAT table. When the return packet arrives, NAT restores the target address to the original internal host using this mapping table, achieving transparent forwarding. There are three types of NAT: **static NAT** (fixed one-to-one mapping), **dynamic NAT** (dynamically allocated from a pool of public IP addresses), and **port address translation (PAT)** (most commonly used, also called NAT overload, mapping multiple internal IPs to the same public IP with different ports). NAT not only saves IPv4 address resources but also provides a degree of isolation and security.

DNS (Domain Name System) is the "phone book" of the internet, used to **translate human-readable domain names (e.g., www.google.com) into machine-readable IP addresses**. It supports a distributed hierarchical structure, including root domains, top-level domains, and authoritative name servers. It is essential for accessing websites, services, and APIs. A device providing public services typically undergoes DNS resolution and NAT mapping to be accessible by end-users.

IP Datagram is the basic unit of data transmitted at the network layer, used to transfer data from the source address to the destination address in the network. It consists of two parts: the **header** and the **payload**. The **header** contains **essential information**, such as **source IP, destination IP, TTL (Time to Live), protocol type (e.g., TCP/UDP), data length, fragmentation flags**, etc., for addressing, routing, and control in the network. The **payload** contains the **actual data** to be transmitted, such as a TCP or UDP segment. IP datagrams are **connectionless and unreliable**, meaning they may be delivered out of order, dropped, or duplicated. Therefore, a transport layer protocol like TCP is needed to ensure reliability. In IPv4 networks, a large data payload may be fragmented into multiple IP datagrams, with the target device reassembling them. Understanding the structure of IP datagrams is fundamental to mastering network layering and data forwarding mechanisms.

VPN (Virtual Private Network) is a technology used to create an "encrypted private tunnel" over a public network. Its core purpose is to provide secure, encrypted, and authenticated remote communication in insecure network environments. VPN encapsulates and encrypts data at the network or transport layer, allowing users or devices at different locations to communicate **as if they were on the same private network**, thus ensuring confidentiality and integrity during data transmission.

4. TCP Implementation, Slow Start, Selective ACK, L2/L3 Load Balancing Strategies, VRRP (Virtual Router Redundancy Protocol)?

TCP is a connection-oriented, **reliable transmission protocol** that establishes a connection through a **three-way handshake** and **releases** the connection through a **four-way handshake**. It uses sequence numbers, acknowledgment numbers, and window size to control reliable data transmission and handles packet loss through retransmission mechanisms. It also includes congestion control (such as slow start, congestion avoidance, fast retransmit, and fast recovery) and flow control mechanisms to ensure reasonable network load. Different operating system kernels (e.g., Linux) implement the TCP protocol stack in kernel space, managing connection states, sliding windows, timers, and other core components, with users completing data transmission via the socket interface.

Slow Start is the first phase of **TCP congestion control**, used to quickly probe the network bandwidth in the early stages of a connection. TCP’s initial send window is typically 1 MSS (Maximum Segment Size), and for every received ACK, the window size doubles, leading to exponential growth until the slow start threshold (ssthresh) is reached or packet loss occurs. Upon packet loss, TCP reduces the window size and enters the congestion avoidance phase, where the window grows linearly. Slow Start avoids overwhelming the network with excessive data at the start of the connection, balancing TCP performance and stability.

Selective Acknowledgment (SACK) is an extension mechanism in TCP to more efficiently handle packet loss scenarios. Traditional ACK can only confirm the last received byte in a continuous stream, while SACK allows the receiver to tell the sender which segments were received and which were lost, enabling the sender to retransmit only the missing parts. This greatly improves transmission efficiency in high-latency or high-loss networks, particularly for wireless networks or long-link transmission. SACK is enabled via the TCP option field and is supported by most modern operating systems by default.

Load balancing can be divided into **L2, L3, L4, and L7** strategies according to the OSI model. **L2 load balancing** forwards based on **MAC** addresses and is commonly found in hardware-based **switched networks**, suitable for high-performance local data center forwarding. **L3 load balancing** operates at the **IP layer**, employing strategies like **ECMP** (Equal-Cost Multi-Path) and **NAT** (Network Address Translation) to distribute traffic and is the basis for high-performance software load balancing, ideal for large-scale service distribution. Compared to application-layer **L7 load balancing**, L2/L3 load balancing is more lightweight, has lower latency, and is commonly used in network core or forwarding layers.

VRRP (Virtual Router Redundancy Protocol): provides high availability by assigning a virtual IP shared among routers. If master fails, backup takes over seamlessly.

5. Cryptography, symmetric encryption, asymmetric encryption, differences?

Cryptography is the study of protecting the confidentiality, integrity, authenticity, and non-repudiation of information. Core methods include encryption algorithms, hash functions, and signature mechanisms. Encryption can be divided into **symmetric encryption** and **asymmetric encryption**. **Symmetric encryption** (e.g., **AES**) uses the **same key for both encryption and decryption**, making it fast and suitable for encrypting large data. **Asymmetric encryption** (e.g., RSA, ECC) uses a **public key for encryption and a private key for decryption**, making it ideal for key exchange and digital signatures. **Hash algorithms** (e.g., **SHA-256**) are irreversible and are used to verify data integrity, while **digital signatures** combine hashing and asymmetric encryption to verify the origin of messages and ensure the content has not been tampered with. In practical applications, HTTPS uses the TLS protocol to exchange session keys via asymmetric encryption and then uses symmetric encryption to transmit data, balancing security and efficiency.

Symmetric encryption uses the **same key for both encryption and decryption**. The key must be securely shared between parties before communication, and typical algorithms include **AES** and **DES**. Symmetric encryption is fast and well-suited for encrypting **large amounts of data**, such as **files or videos**. However, key distribution is a challenge, as if the key is leaked, security is compromised.

Asymmetric encryption uses a pair of keys: **a public key for encryption and a private key for decryption**. The public key can be shared, while the private key remains confidential. It is commonly used for identity verification, key exchange, and digital signatures, with typical algorithms including **RSA and ECC**. While asymmetric encryption is secure and facilitates easy key management, it is slower and not suitable for encrypting large amounts of data. Therefore, it is commonly used in combination with symmetric encryption (e.g., in TLS, where asymmetric encryption is used to exchange symmetric keys, and symmetric encryption is then used to transmit data).

In summary: **Symmetric encryption** is fast but faces challenges in key distribution; **Asymmetric encryption** is secure but slower, commonly used for key exchange and authentication.

6. How to troubleshoot latency or connectivity issues in network services? Tools?

Start by using **ping** to check if the target host is reachable, and use **traceroute** or **mtr** to trace the route and identify which hop has high network delay or packet loss. Use **tcpdump** or **Wireshark** to capture packets and analyze TCP three-way handshakes, retransmissions, etc. For checking server port usage, use **netstat** or **ss**, and simulate requests using **curl** to check the response status. **lsof** can help check if ports are being occupied or if there are file handle leaks. By combining these tools with logs, you can layer down to pinpoint the cause of network anomalies.

7. How to ensure stable video stream delivery?

Ensuring stable video stream transmission requires optimizing several aspects, including the **transmission protocol, buffer control, packet loss recovery, and bitrate adaptation mechanisms**. At the lower layers, **UDP** is typically chosen for the **transmission** protocol, combined with **RTP** to achieve ordered transmission and timestamp synchronization. **FEC** (Forward Error Correction) or **NACK/SRTP** can be used for **packet loss recovery** to reduce retransmission delays. At the **transport** layer, **adaptive bitrate (ABR)** is used to automatically adjust video quality based on available bandwidth, avoiding playback stuttering. **Client-side buffer control** (e.g., setting reasonable buffer sizes and low-latency pre-loading) ensures smooth playback. For cross-network transmission, **QUIC** or custom **UDP with congestion control protocols** can reduce connection setup delays and improve jitter resistance. In general, stable video streaming relies on the collaborative work of "lightweight protocols + elastic jitter tolerance + intelligent adjustment + network status monitoring" to provide a low-latency, high-availability video experience.

8. How to debug encrypted data (packets, streams, etc.)?

Debugging encrypted information (such as **encrypted packets or video streams**) essentially revolves around "**ciphertext observability + plaintext recoverability**." The typical approach is to **configure decryption keys** in development or testing environments, disable TLS, or instrument logs to observe intermediate states before and after encryption. For **TLS/SSL** encrypted communication, packet capture tools like **Wireshark** combined with **pre-master keys** (via the SSLKEYLOGFILE) can be used to decrypt and view plaintext. **For video streams (such as AES-encrypted H.264/TS streams)**, tools like **FFmpeg + Hex Editor** can be used with known keys for decryption and analysis. **For custom protocol encryption**, **logging encryption/decryption function inputs and outputs, or caching plaintext before encryption, can be helpful**. Additionally, when debugging on **devices**, **serial/ memory-mapping/DMA trace tools** can be used to observe memory regions before and after encryption. The goal of debugging is to verify "whether encrypted data is correct, whether decryption restores the plaintext, and whether the keys are agreed upon." Detailed logging of critical parameters such as symmetric keys, random number seeds, and IVs may be required for replay testing.

9. What happens when you type a URL and press Enter in a browser?

When the user types a **URL** in the **browser** and presses Enter, the browser first checks if it can find the **relevant DNS, HTTP, or page caches**. If not, it initiates a **DNS** query to resolve the **domain name to an IP address**. The **browser** then establishes a **TCP connection** via a three-way handshake (if HTTPS, a TLS handshake is also performed to encrypt the communication). **Next**, the browser **constructs and sends** an **HTTP request**. The server receives the request, generates a response, and **returns HTML content**. After receiving the response, the browser parses it and starts the rendering process, which includes parsing the HTML to construct the **DOM tree**, parsing CSS to construct the **CSSOM**, merging them into a **render tree**, calculating layout (**Reflow**), and drawing (**Paint**). If JavaScript is present, it is executed by the JavaScript engine and may trigger reflow or repaint through DOM manipulation. Meanwhile, images, fonts, and other resources are loaded asynchronously, and finally rendered on the page. This entire process involves various components working together, such as network transmission, protocol stack, browser engine, multi-process models, and rendering engines.

10. How does a router assign an IP address?

When a device connects to a **router** (e.g., via Ethernet or Wi-Fi), the network card first initializes and starts network services. It then broadcasts a **DHCP** Discover message to request a dynamic IP address. The router, acting as a DHCP server, receives the request and replies with a DHCP Offer, containing available IP address, subnet mask, gateway, DNS, and other information. The client then sends a DHCP Request indicating acceptance of this configuration, and the router confirms with a DHCP ACK. After these four interactions, the device successfully obtains the dynamically assigned IP address and network parameters, such as the gateway. The device then configures its local network stack, updates the routing table, and binds the network interface, enabling it to access the local network or the internet via the router. This process is based on the **DHCP (Dynamic Host Configuration Protocol)**, which is the standard procedure for automatic networking, and it is the most common method of IP address allocation between modern routers and clients.

11. How to design a gateway system that can defend against DDoS attacks?

Designing a gateway system that can defend against DDoS attacks requires considering multiple aspects, including traffic characteristic identification, resource isolation, dynamic policies, and traffic cleaning mechanisms.

First, deploy firewalls or gateways with rate-limiting features at the network entrance. Set connection count thresholds and rate limits for common attack patterns such as TCP SYN floods, UDP floods, and HTTP floods, and enable defenses like SYN Cookies to prevent the exhaustion of half-connection resources.

Next, introduce reverse proxies and caching mechanisms to handle static requests at the edge, reducing the load on the origin server. For anomaly traffic detection, combine real-time log analysis, IP clustering, and behavioral pattern recognition, and dynamically adjust blocking strategies or enable blacklist/whitelist mechanisms to ensure that legitimate traffic is not mistakenly blocked. For large-scale attack scenarios, the system should have cloud-cleaning capabilities or BGP traffic diversion to direct abnormal traffic to powerful DDoS protection platforms for filtering.

Additionally, establish a comprehensive monitoring and alerting system to monitor key indicators such as connection counts, CPU usage, and traffic fluctuations in real-time, triggering automated responses. By embedding defense strategies into the system architecture and creating an integrated "Perception-Defense-Response" loop, the gateway system can effectively withstand DDoS attacks.

12. Principles and differences between Firewalls and Intrusion Detection Systems (IDS)?

Both **Firewalls** and **Intrusion Detection Systems (IDS)** are crucial network boundary defense devices, but they differ in their design objectives and operational mechanisms. A **firewall**'s core function is "access control." It operates at key points in the network, using rule matching to determine which connections are allowed and which are blocked. Its decisions are based on factors such as IP, port, protocol, or application-layer characteristics, making it an active defense mechanism.

In contrast, **IDS** focuses on "passive detection." It does not control whether traffic passes through, but instead analyzes incoming and outgoing traffic in real time without interfering with the network communication. IDS looks for known attack signatures or suspicious behaviors, logs anomalies, and triggers alerts when irregularities are detected. A firewall is more like a gatekeeper, while an IDS acts as a scout. They are often deployed together, forming an effective defense through a combination of "Control + Perception." In practice, firewalls are typically used to intercept northward traffic, while IDS can analyze both north-south and east-west lateral traffic, and their combined use significantly improves overall network security.

13. Typical Deployment Methods of IDS and IPS?

IDS (Intrusion Detection Systems) and IPS (Intrusion Prevention Systems) are very similar in functionality, both based on deep packet inspection for threat identification. The key difference is whether they possess "real-time blocking capability."

IDS systems are usually deployed in a bypass listening mode, where they copy network traffic via a mirroring port (SPAN) or TAP devices for analysis. Since they don't directly interfere with production traffic, IDS is more suitable for high real-time or high-availability environments.

In contrast, **IPS** is deployed directly in the traffic path as a "serial device." When it detects an attack, it immediately drops packets, redirects traffic, or blocks the source IP. IPS is suitable for network cores, server front-end, application boundary, and other critical locations. In practice, IDS is used more for detection and alerting, often integrated with SIEM or situational awareness platforms for further analysis. IPS, on the other hand, acts as the first line of defense at the network perimeter or DMZ, providing immediate responses to known threats. Modern systems often integrate IPS modules into firewalls, offering unified boundary security capabilities.

14. How to implement an anomaly detection module based on IP + behavior patterns?

An **IP and behavior pattern-based anomaly detection module** primarily analyzes the source IP address and its request behaviors in network traffic, using machine learning or rule engines to identify potential malicious activities. First, the system collects behavior data for each IP, such as access frequency, requested resource types, and connection duration. Then, a normal traffic model is built based on historical traffic data, and threshold detection rules are established. For example, if a single IP frequently accesses the same resource within a short period or makes excessive requests to different resources, these actions might be flagged as abnormal behavior.

Additionally, behavior pattern detection can be enhanced by considering user device information (such as User-Agent, geographic location, etc.) and using data mining techniques like clustering analysis or outlier detection algorithms to identify new attack patterns. This approach allows the system to accurately detect common network attacks such as brute-force cracking, crawler attacks, and information leakage, triggering alerts or automatically blocking the offending IP address.

15. How to implement a blocking system with automatic blacklist updates?

A **blocking system with automatic blacklist updates** needs to support automated monitoring, real-time updates, and efficient blocking. First, the system should integrate a data source to automatically obtain newly identified malicious IP addresses or domain names. These data sources can come from network traffic analysis, third-party threat intelligence platforms, user feedback, or community contributions. By continuously monitoring network traffic and comparing it with existing blacklists, the system can quickly identify and record malicious IPs. When a new malicious IP is detected, the system automatically adds it to the blacklist and immediately applies blocking policies on firewalls, IPS, or other security devices.

To avoid false positives, the system should also include a whitelist mechanism to ensure trusted IPs are not mistakenly blocked. Additionally, the frequency of blacklist updates should be adjusted based on attack activity and changes in the network environment, ensuring timely attack responses while minimizing performance issues caused by frequent updates. Through a combination of automated updates and real-time blocking policies, the system can effectively prevent DDoS, crawling, brute-force cracking, and other attack types.

16. Principles and Performance Optimization of DPI (Deep Packet Inspection) Technology

DPI (Deep Packet Inspection) technology involves analyzing data packets transmitted over a network, extracting their internal content to inspect various protocols, application-layer data, and potential security threats. Unlike traditional packet filtering, DPI can analyze packets at a higher level, identifying not just header information but also the data payload. DPI is widely used in traffic analysis, virus detection, spam filtering, and enhancing firewall policies. Its core principle is to deeply analyze each data packet, matching it against predefined rules or signatures to check for malicious code, sensitive data, or abnormal behaviors.

For performance optimization, DPI must handle high throughput, so technologies like hardware acceleration (e.g., **FPGA**, **ASIC**), multi-threaded processing, and traffic diversion are often employed to speed up processing. Additionally, traffic sampling and protocol recognition can help optimize by reducing the processing of irrelevant data, preventing unnecessary computation resources from being wasted on invalid packets. In practical deployments, DPI may also utilize a distributed architecture for traffic processing, ensuring real-time detection of large-scale network traffic and safeguarding network security and service quality.

17. What's the difference between Authentication and Authorization? What are the principles behind protocols such as SAML, LDAP, RADIUS, TACACS+, and 2FA?

Authentication is the process of identifying a user, typically by **verifying credentials** such as **username/password, tokens, fingerprints, or smart cards**—it answers the question "**Who are you**?". **Authorization** occurs after authentication and determines what resources the user can access **based on roles and permissions**—it answers "**What can you do**?".

SAML is an **XML**-based cross-domain authentication protocol that enables Single Sign-On (SSO) using browser redirection and signed tokens. It's widely used in enterprise integration with cloud platforms (e.g., logging into Salesforce or AWS).

LDAP is a hierarchical directory access protocol used for centralized identity and group management. It supports operations like bind for **credential verification**.

RADIUS combines authentication and authorization in a single UDP request, commonly used in network access scenarios like VPN, Wi-Fi, and NAS, with message integrity ensured by a shared secret.

TACACS+ separates Authentication, Authorization, and Accounting (AAA), uses TCP for transmission, and encrypts the entire payload, allowing fine-grained command-level access control—typically used in managing access to network devices.

EAP is an extensible authentication framework under 802.1X, supporting methods like EAP-TLS, EAP-TTLS for secure device authentication.

2FA (Two-Factor Authentication) introduces an extra layer of identity proof, usually via TOTP (e.g., Google Authenticator), SMS codes, or hardware tokens (e.g., YubiKey), and is often integrated into OAuth2 flows, RADIUS logins, or Web-based authentication portals to improve security.

18. How do firewalls enforce user-based authentication and access control?

Modern firewalls support user-identity-based policy enforcement by integrating with identity services such as **LDAP, RADIUS, or local databases**. Upon **first** access, users may be prompted to **authenticate** via captive portal, 802.1X, or VPN login. Once authenticated, their identity is **mapped to an IP address**, MAC address, or session. The firewall policy engine can **then apply rules based on combinations of user group, application, destination address, and service port for fine-grained traffic control**.

For inline **authentication**, firewalls may **cache user sessions, maintain state using cookies or tokens**, and support features like idle timeout, forced re-authentication, or adaptive policies based on geolocation and device fingerprinting. When integrated with Active Directory or SSO servers, firewalls can transparently detect identity, enrich log data with user context, and improve auditing, forensics, and access analytics.

19. How is customized TLS communication established between devices and cloud services?

To secure and control communication between a device and its cloud server or API endpoint, standard TLS can be extended with several customizations. This may include using certificates signed by trusted CAs or internal PKI, or enabling certificate pinning to prevent man-in-the-middle (MITM) attacks.

Mutual TLS (mTLS) may be used, requiring the client to present its certificate for identity verification. During the handshake, the cipher suites and TLS versions can be customized (e.g., enforcing TLS 1.3 and disabling vulnerable ciphers like CBC or RC4). Application-layer protocol negotiation (ALPN) can be used to specify the upper-layer protocol such as HTTP/2 or MQTT.
To convey identity after the handshake, JWT tokens, OAuth2 tokens, or custom headers (like device IDs) can be injected. TLS session reuse via session tickets or session IDs can improve performance.

In real deployments, further considerations include handshake timeout handling, certificate renewal and rotation, connection retry logic, and fallback strategies (e.g., disable HTTP downgrade, enforce strict TLS-only communication).

20. How should multiple network modules securely communicate with Fortinet or other cloud servers? What boundaries should be respected?

In systems composed of multiple network modules—for tasks like authentication, config fetching, state reporting, and log forwarding—secure communication channels and clear separation of concerns are essential. TLS must be uniformly implemented across modules, with centralized certificate and key management. Identity verification can be performed using OAuth2 tokens, HMAC signatures, or other schemes.

The authentication module may handle token retrieval and caching (e.g., using Redis with expiration and renewal). The configuration module should support robust JSON/YAML parsing and validation. Logs may be buffered and forwarded asynchronously via Kafka or Fluentd, and heartbeat or event-based status updates can be managed by dedicated synchronization modules.
Special attention should be paid to token leakage between modules, TLS session hijacking, and configuration tampering. It is recommended to encapsulate a secure connection pool and key management module to isolate sensitive data from business logic.

To support reliability in weak network environments, fallback mechanisms should be in place, including reconnection logic, local cache utilization, and certificate failure handling—ensuring devices can continue functioning even if the cloud becomes temporarily unavailable.

Algorithm and Data Structure

1. Common Sorting Algorithms

Sorting Algorithm	Best Case	Average Case	Worst Case	Space Complexity	Stable
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes
Selection Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	No
Shell Sort	O(n log n)	Depends on increment sequence	O(n^2)	O(1)	No
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)	No
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	No
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(k)	Yes
Bucket Sort	O(n + k)	O(n + k)	O(n^2)	O(n + k)	Yes
Radix Sort	O(nk)	O(nk)	O(nk)	O(n + k)	Yes

2. Common Algorithm Paradigms and Applications

Paradigm	Typical Applications	Complexity / Feature	Notes / Suggestions
Divide and Conquer	Merge Sort, Quick Sort, Matrix Multiplication	Recursive split + merge, O(n log n)	Best for problems that can be broken down evenly
Greedy Algorithm	Activity selection, interval scheduling, Huffman Coding	Local optimal → global optimal	Requires greedy-choice property, counterexamples important
Dynamic Programming	Knapsack, Edit Distance, Interval DP, Stock Trading	O(n^2)~O(n^3), compressible space	State definition & transition are key
Bit Manipulation	Subset enumeration, bitmask DP, sieve optimization	O(1) per op, compact space	Useful for boolean problems, subset tricks
Graph Algorithms	Dijkstra, BFS/DFS, TopoSort, Tarjan, MST	Depends on graph size	Model dependencies, paths, connectivity
String Algorithms	KMP, Z-Algorithm, Trie, Aho-Corasick	O(n + m)	Used in pattern match, filters, prefix search

Search Algorithms	DFS, BFS, A*, Bi-BFS, Iterative Deepening DFS	Guided search depth/heuristic	Core for pathfinding and traversal
Number Theory	Fast Power, GCD, Euler's Phi, Linear Sieve, ExGCD	Mostly O(log n)	Used in combinatorics, crypto, modular math

3. Common Data Structure and STL

std::vector: A dynamic array with **O(1) random access**. push_back() is O(1) when the capacity is sufficient and amortized O(1) when expanding. **insert() and erase() are O(n) for non-tail operations**. Suitable for dynamically changing elements, with frequent access and tail-end operations, such as dynamic caches, pagers, and path records. In C, dynamic arrays can be implemented using malloc + realloc. In Java, the equivalent is ArrayList.

std::list / std::forward_list: Doubly linked list and singly linked list, respectively, that support **O(1) insertion and deletion at any position but do not support random access**, making lookups O(n). Common functions include push_front(), insert(), and erase(). Suitable for scenarios with frequent insertions/deletions but no need for indexing, such as task schedulers, history records, and LRU cache linked list parts. In C, linked lists can be manually implemented using structs and pointers.

Comparison	vector (dynamic array)	list (doubly linked list)	Usage Suggestion
Underlying Structure	Contiguous array	Pointer-linked nodes	vector for dense access, list for frequent inserts/deletes
Random Access	O(1)	O(n)	vector for index access
Insert/Delete	Middle: O(n), End: O(1)	O(1) (after locating node)	list for frequent mid ops, vector faster at end
Memory Efficiency	High	Low (extra pointer space)	vector more space-efficient
Cache Locality	Good	Poor	vector better for performance-

			critical tasks
Traversal Speed	Fast	Slow	Prefer vector for large sequential traversal
Use Cases	Stack, arrays, sliding window	LRU cache, list problems	Choose by algorithm nature

std::stack / std::queue / std::deque: Stack and queue containers are typically based on deque. **push() and pop() are O(1), and top() and front()/back() access are also O(1)**. Suitable for expression evaluation, message buffers, and breadth-first searches. In C, stacks are implemented using arrays or linked lists, while cyclic queues or linked queues can be used for queues. In Java, ArrayDeque replaces traditional Stack, and LinkedList implements the Queue interface.

Comparison	stack (adapter)	deque (double-ended queue)	Usage Suggestion
Underlying Structure	Based on deque	Native two-end ops	stack is simplified wrapper, fewer functions
Functionality	push/pop/top only	front/back, insert/delete	Use deque for richer features
Adapter Trait	No iteration	Random access supported	deque for container-style ops
Use Cases	Function stack, bracket match	Sliding window, LRU cache	Prefer deque for 2-end operations

std::priority_queue: Implemented as a max heap. **push() and pop() are O(log n), and top() is O(1)**. Internally uses std::vector with push_heap and pop_heap for implementation. Suitable for scheduling tasks, greedy algorithms, and shortest path algorithms (e.g., Dijkstra). In C, a max heap can be manually implemented using arrays, and in Java, PriorityQueue (which defaults to a min-heap) can be customized with a comparator to implement a max-heap.

std::unordered_map / std::unordered_set: Based on hash tables**, supporting average O(1) insertions, lookups, and deletions, with the worst case being O(n)**. Common functions include find(), insert(), and erase(). **Suitable for frequent lookups and fast** location-based operations, such as frequency counting, cache mapping, and de-duplication. In C, hash tables are manually implemented, requiring a hash function and collision resolution (e.g., chaining or open addressing). In Java, the equivalent is HashMap and HashSet, with JDK 8+ automatically converting to red-black trees when hash bucket lengths exceed a threshold.

std::map / std::set: Based on red-black trees, **insertion, deletion, and lookup are all O(log n)**, supporting ordered traversal and range queries (e.g., lower_bound, upper_bound). Suitable for scenarios that **require sorting and range searching**, such as ranking systems and interval merging. In C, red-black trees can be manually implemented or using third-party libraries like libavl. In Java, TreeMap and TreeSet use red-black trees internally to maintain ordered keys.

Comparison	map (Red-Black Tree)	unordered_map (Hash Table)	Usage Suggestion
Lookup Time	O(log n)	O(1) average	unordered_map for performance-critical lookup
Ordering	Key-sorted	Unordered	Use map if sorted/range lookup needed
Memory Usage	Compact	Larger (hash buckets)	map for memory-sensitive usage
Iteration Order	Sorted by key	No order	Use map when order matters
Insert/Delete	O(log n)	O(1) average	unordered_map better for frequent inserts
Use Cases	OrderedMap, interval trees	Hash map, frequency counting	unordered_map more efficient in counting tasks

Comparison	set (Red-Black Tree)	unordered_set (Hash Table)	Usage Suggestion
Lookup Time	O(log n)	O(1) average	unordered_set for

			faster search
Ordering	Ordered by key	Unordered	set for sorted output
Duplicate Keys	Not allowed	Not allowed	Use multiset for duplicates
Memory Usage	Compact	Larger (hash buckets)	set more memory friendly
Iteration Order	Sorted	Unordered	set when order matters
Use Cases	Sorted unique elements	Fast deduplication & lookup	unordered_set efficient for frequency use

Heap(make_heap, push_heap, pop_heap): C++ STL provides heap operations, with heap construction being O(n) and insertion/deletion O(log n). Suitable for Top-K problems, priority schedulers, and greedy algorithms. In C, heap structures are manually maintained using arrays, and in Java, PriorityQueue defaults to a min-heap but can be extended to a max-heap.

Union-Find: Through path compression and union by rank, the find and union operations have time complexity close to O(1), specifically O(α(n)), where α is the inverse Ackermann function. In C++, it's commonly implemented using vector and recursive path compression. In C, integer arrays manage parent and rank. Java implementations usually involve int[] parent and int[] size/rank.

Trie: Used for efficient string set operations, **insert and lookup are O(L), where L is the string length**. Suitable for prefix matching, autocomplete, and search suggestions. In C++, node structures can be implemented with map or arrays. In C, static allocation is often implemented using structs and character arrays. In Java, TrieNode classes with HashMap or arrays of length 26 are used.

Structure	Typical Usage	Complexity	Recommendation
Priority Queue (Heap)	Top-K, Scheduler, Shortest Path	Insert/Delete: O(log n)	Use std::priority_queue for dynamic max/min tasks
Union Find (DSU)	Connectivity,	O(α(n)) nearly	Supports path compression, use in

	Network Merge	constant	graph connectivity
Trie	Prefix, Autocomplete, Word Count	O(m), m = string length	Efficient for batch word lookup
Deque	Sliding Window, LRU cache	Insert/Delete: O(1) at both ends	More flexible than stack/queue
Graph (Adj List)	Shortest Path, Components	Build/Traverse: O(n + e)	Model complex relations, use custom or Boost.Graph

Skip List: A probabilistic balanced list structure with average **insertion, deletion, and lookup times of O(log n) and worst case O(n).** It is simple to implement and suitable as a replacement for balanced trees. **Redis' ordered sets use skip lists**. In both C++ and Java, manual multi-level linked list logic is required; C implementations can reference Redis' source code for guidance.

Bloom Filter: A space-efficient probabilistic structure used for set membership queries with tolerable false positives. **Insertion and lookup operations are O(k),** where k is the number of hash functions. Deletions are not supported. Commonly used for cache penetration defense, uniqueness checking in big data, and blacklisting. In C/C++, it is implemented using bit arrays and multiple hash functions; Java has libraries like Guava for quick implementation.

LRU Cache (list + unordered_map): A doubly linked list maintains access order, while a hash map maps keys to nodes, with all operations being O(1). Suitable for **cache eviction strategies, database page caching, and browser caching**. C++ uses a combination of list and unordered_map. In C, a custom doubly linked list and hash table (e.g., using uthash) are used. In Java, LinkedHashMap can be subclassed to override removeEldestEntry.

Segment Tree: Supports O(log n) range queries and point/interval updates, with nodes maintaining sum, min/max, lazy propagation, etc. C++ and C often implement it using arrays to simulate tree structures with both recursive and iterative tree-building methods. In Java, it is often encapsulated in class structures and is ideal for interview scenarios involving range query problems.

Structure	Typical Usage	Complexity	Recommendation
Skip List	Ordered Map with concurrency support	O(log n)	Alternative to trees, used in Redis, better concurrency
Segment Tree	Range query and updates (e.g., RMQ)	Build: O(n), Query/Update: O(log n)	Supports lazy propagation for efficient range ops
Fenwick Tree (BIT)	Prefix sum, inversion count	Update/Query: O(log n)	Compact for 1D prefix sums
Sparse Table	Static RMQ (min/max)	Preprocessing: O(n log n), Query: O(1)	Use for static range queries
Persistent Segment Tree	Versioned queries/snapshots	Each update: O(log n)	Supports rollback/history query

Database Systems and Data Processing

Keyword

Transaction is the smallest unit of operation in a database that ensures data consistency and integrity. A transaction follows the ACID principles: **Atomicity, Consistency, Isolation, and Durability**. Multiple SQL operations can be grouped into one transaction, which either fully succeeds (commit) or completely rolls back. For example, in an e-commerce system, creating an order, deducting inventory, and debiting an account should all occur within a single transaction to prevent partial updates.

Index is an auxiliary data structure used by databases to accelerate query efficiency, similar to the index of a book. Common types include **B+ Tree** index (default in MySQL), hash index, and full-text index. Indexes reduce the number of scanned rows, greatly improving query speed, but they introduce overhead during inserts and consume additional space. Commonly indexed fields include order numbers, user IDs, and timestamps.

Normalization is a methodology in relational database design aimed at reducing redundancy and improving consistency. Common normal forms include 1NF (atomic fields), 2NF (full dependency on the primary key), and 3NF (removal of transitive dependencies). While normalization improves consistency, over-normalization can affect performance. In practice, denormalization is sometimes adopted for performance, such as duplicating fields to speed up query efficiency.

Primary Key / Foreign Key are essential constraints in relational databases to maintain inter-table integrity. A primary key uniquely identifies each row and must not be duplicated. A foreign key references another table's primary key and defines the relationship between tables. For instance, in a user-order relationship, the user_id in the order table is a foreign key pointing to the user table.

Isolation Level defines the visibility of data changes between concurrent transactions. SQL standard defines four levels:

Read Uncommitted is the lowest isolation level in databases. It allows a transaction to read uncommitted changes made by other transactions, which can lead to **dirty reads**. In other words, a transaction may access data that is still in the process of being updated by another transaction, and if those changes are later rolled back, the read result would be invalid or incorrect. This severely compromises data consistency. For example, a user querying their account balance might see a temporary value that is ultimately rolled back, leading to a mismatch between what is displayed and what actually exists. Although this level provides minimal locking overhead and the highest concurrency performance, it offers extremely poor data reliability and is generally not suitable for production environments. It is only applicable in scenarios with extremely low consistency requirements and a focus on read speed, such as real-time log analysis, monitoring data collection, or database debugging. Mainstream databases like MySQL, PostgreSQL, and Oracle **do not use this level by default** and typically require explicit configuration to enable it, making it rarely used in actual development. **Dirty read** refers to a transaction reading uncommitted changes made by another transaction. If that data is later rolled back, the reading transaction ends up with an invalid or incorrect result. Dirty reads typically occur under the **Read Uncommitted** isolation level. For instance, if one user is modifying an account balance and another reads the uncommitted value, it can lead to incorrect prompts or calculation errors.

Read Committed is the **default isolation level in most databases**, such as **Oracle, PostgreSQL, and SQL Server**. It ensures that a transaction can only read data that has been committed by other transactions, thereby effectively preventing dirty reads. However, this level can still result in **non-repeatable reads**, meaning that if a transaction reads the same record multiple times, and another transaction modifies and commits changes to that record in the meantime, the results of each read may differ. For example, in an **order management system, a user queries the status of an order twice within the same transaction**. If another user updates and commits the order status between these two queries, the results will be inconsistent, potentially causing business logic errors. Despite this consistency risk, Read Committed strikes a good balance between performance and data reliability and is therefore widely used in scenarios involving frequent write operations and moderate consistency requirements, such as order processing in e-commerce, content editing workflows in CMS systems, and general administrative interfaces.

Repeatable Read is the default isolation level for the InnoDB storage engine in **MySQL**. It ensures that if a transaction reads the same record multiple times, the result remains consistent throughout the transaction, thereby preventing dirty reads and non-repeatable reads. However, it may still allow **phantom reads**, where repeated queries within the same transaction return different result sets due to new rows being inserted by other transactions that meet the original query criteria. MySQL mitigates this issue using **Multi-Version Concurrency Control (MVCC)** and **gap locking**, which helps to a certain extent in preventing phantom reads. Repeatable Read is suitable for scenarios **requiring higher data consistency, such as inventory validation, financial reporting, and statistical computations**. It ensures that a consistent snapshot is maintained throughout the transaction. Compared to Read Committed, it provides stronger consistency at the cost of slightly reduced concurrency performance, making it ideal for applications involving financial records, account balances, and other cases where precise data reads are critical.

Serializable is the **highest isolation level** available in relational databases. It prevents all types of concurrency anomalies, including dirty reads, non-repeatable reads, and phantom reads. It enforces strict transaction ordering, typically through locking mechanisms or conflict detection strategies, making concurrent transactions behave as if they were executed sequentially. While this ensures the highest level of data consistency and integrity, it significantly reduces concurrency, potentially leading to transaction queuing, increased chances of deadlocks, and degraded overall throughput. Serializable is best suited for mission-critical applications where absolute accuracy is required, such as **bank transfers, financial settlements, and monthly accounting operations**. In PostgreSQL, it is implemented as **Serializable Snapshot Isolation (SSI)**, which provides better performance through conflict detection instead of strict locking. In Oracle and SQL Server, it can also be enabled explicitly. Despite its reliability, due to its performance cost, Serializable is rarely used in routine business applications and is typically reserved for high-sensitivity, high-value, and high-risk transactional logic.

Locks are mechanisms used to manage concurrent access to data, including row-level locks, table-level locks, intent locks, shared locks, and exclusive locks. Locks prevent data conflicts and ensure consistency. Proper use of locks (e.g., avoiding long transactions or full table scans) is crucial for performance in high-concurrency environments.

View is a virtual table derived from a SQL query that does not store actual data. Views can encapsulate complex logic, hide sensitive fields, and standardize access to data. For example, a view can merge user profiles and user levels into a single read-only interface for front-end queries.

Trigger is a predefined operation that automatically executes in response to specific actions like INSERT, UPDATE, or DELETE. Triggers are useful for validation, logging, and auditing. For instance, when an order's status is updated to "shipped," a trigger can insert a record into the shipping log table.

Stored Procedure / Function are precompiled SQL statement sets used to encapsulate business logic and improve execution efficiency. Stored procedures can contain control logic and transactions, while functions return calculated results. They are often used for database migration, access control, or business encapsulation.

Join is a key SQL operation for combining data from multiple tables. Common join types include INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL JOIN. Joins are essential for data integration, such as displaying orders with user details on the front-end.

Pagination is used to limit the number of records returned in a query. SQL syntax includes LIMIT/OFFSET (MySQL, PostgreSQL) or TOP/FETCH NEXT (SQL Server). Pagination improves API responsiveness and is widely used in e-commerce systems for product listings and search results.

Aggregation Functions are SQL functions used for statistical summaries, including COUNT, SUM, AVG, MAX, and MIN. They are commonly used with GROUP BY to generate reports, such as monthly order counts or user distribution by region.

Temp Tables / CTE (Common Table Expression) are temporary structures used for intermediate calculations. Temp tables are session-scoped and useful for breaking down complex logic. CTEs (WITH clause) improve query readability and support recursion. They're widely used in reporting, tree traversal, and data transformation tasks.

Execution Plan is the execution strategy tree generated by the database after parsing a SQL query. It outlines the access path, index usage, and join method. Tools like EXPLAIN (MySQL) help developers analyze and optimize SQL performance based on the plan.

1. What is a database? What types of databases are there? What are the differences between PostgreSQL, ClickHouse, and MySQL?

A database is a software system used for storing, managing, and querying structured or semi-structured data, supporting efficient data insertion, retrieval, updates, and deletion.

Databases can be broadly classified into several categories: the most common are relational databases (**RDBMS**), such as **MySQL and PostgreSQL**, which organize data in tables and support SQL queries; non-relational databases (**NoSQL**), such as **MongoDB** (document-based) and **Redis** (key-value), which are suitable for highly scalable and flexible data structures; and **columnar databases like ClickHouse and Apache Cassandra**, designed for analytical and **OLAP** (Online Analytical Processing) tasks, which store data in columns to enhance query performance.

PostgreSQL is a powerful open-source relational database that supports **standard SQL**, complex queries, transactions, and extensions. **ClickHouse**, on the other hand, is a high-performance columnar database that excels at **large-scale data** analysis tasks, offering extremely **fast multi-dimensional aggregation and distributed querying**. It is commonly used in log analysis, metrics calculation, and other scenarios.

Both PostgreSQL and MySQL are widely used open-source relational databases, but PostgreSQL emphasizes standardization, extensibility, and complex querying capabilities, while **MySQL** is known for its **lightweight, easy-to-use nature**, and **high read/write performance**. **PostgreSQL** fully adheres to SQL standards, supporting complex **transactions, window functions**, CTEs (Common Table Expressions), **parallel queries**, GIS, JSON, and other advanced features. It is widely used in financial, telecommunications, government, and research sectors where data consistency and complex computations are critical. Additionally, PostgreSQL has strong extensibility, allowing for custom data types, functions, and index methods, making it suitable for OLAP, BI, and data warehouse systems.

MySQL, in contrast, is simpler to deploy, has a mature community ecosystem, and offers high performance, making it popular in web backends, e-commerce systems, and content management platforms. It is especially common in LAMP (Linux + Apache + MySQL + PHP) architecture. MySQL uses the InnoDB engine by default, which is suitable for transactional processing but lacks flexibility in distributed, high-concurrency, and high-scalability scenarios compared to PostgreSQL. Therefore, when choosing between them, it's important to consider business complexity, data consistency requirements, scalability, and ecosystem dependencies.

2. Why is PostgreSQL suitable for high-security scenarios, and why is MySQL suitable for internet scenarios? How does PostgreSQL handle isolation?

PostgreSQL is more suitable for high-security, high-consistency scenarios because it strictly adheres to SQL standards, has **robust transaction isolation**, and supports multi-version concurrency control (**MVCC**) with **strict enforcement**. It offers strong transaction mechanisms (such as full ACID support and serializable isolation levels), extensive permission management (based on roles, schemas, column-level control), and supports logical replication, WAL (Write-Ahead Logging) archiving, and auditing extensions. This makes it ideal for environments with high requirements for data consistency and compliance, such as finance, government, and research.

MySQL, in contrast, is designed to be **lightweight and performance-oriented**, with a **default isolation level of REPEATABLE READ**, which is simpler to implement and offers high read/write efficiency. It is widely used in internet businesses that involve high read volume, simple business models, and rapid iteration, making it well-suited for web applications, e-commerce, and SaaS platforms. MySQL, when paired with Redis, Elasticsearch, and other technologies, can build high-performance systems, making it a common choice in many web-based applications.

PostgreSQL implements transaction isolation primarily through MVCC and four standard SQL isolation levels to control concurrent read/write behavior and ensure data consistency and transaction isolation. MVCC maintains multiple versions of each data row and uses snapshots in each transaction to read "visible data at the time the transaction started," thus enabling non-blocking reads and avoiding read locks. **PostgreSQL supports four isolation levels: READ UNCOMMITTED (effectively equivalent to READ COMMITTED), READ COMMITTED (default, reads committed data), REPEATABLE READ (ensures repeatable reads based on transaction snapshots), and SERIALIZABLE (the highest level, forcing transaction serialization, potentially through transaction abortion).** At higher isolation levels, PostgreSQL checks for conflicts based on snapshot versions and rolls back transactions to maintain strong consistency. The MVCC design enables PostgreSQL to achieve precise control over isolation policies while maintaining high concurrency performance, making it suitable for business systems with high consistency requirements.

3. What is a slow query? How can PostgreSQL optimize slow queries? How to interpret EXPLAIN output?

A slow query refers to a SQL statement that takes longer than expected to execute and negatively impacts system performance. This often results from full table scans, missing indexes, large data volumes, or inefficient query logic. It is essential to analyze the execution plan, optimize the query, and adjust system resources for performance tuning.

In PostgreSQL, slow queries can be captured by enabling **log_min_duration_statement** or using the **pg_stat_statements** extension. The **EXPLAIN** or **EXPLAIN ANALYZE** commands can then be used to examine the query plan.

The **EXPLAIN** output shows the execution strategy for each node (e.g., **Seq Scan, Index Scan, Nested Loop**), estimated row counts, actual execution times, and other key information. This helps identify whether an index is being used, whether there are inefficient join strategies, or if there are inaccuracies in row count estimates. Common optimization methods include adding appropriate indexes (e.g., BTree, GIN, BRIN), rewriting SQL queries to avoid redundant joins and subqueries, and managing memory used by temporary sorting and hash operations (e.g., work_mem). Query optimization is a critical part of database performance tuning and requires precise analysis based on the business scenario and data distribution.

4. What are the common trees used in databases, and what are their advantages and disadvantages?

The common trees used in databases include **B-Tree, B+Tree, R-Tree, and GiST** (Generalized Search Tree), each serving different data indexing scenarios.

B-Tree: The most classic balanced search tree that supports fast point queries, insertion, and deletion. However, its performance for range queries is slightly lower.

B+Tree: An optimized version of B-Tree where all data is stored in the leaf nodes, and internal nodes only store keys. Leaf nodes are linked together, making range queries more efficient. This is the most common indexing structure in relational databases, such as PostgreSQL and MySQL.

R-Tree: Suitable for storing multi-dimensional spatial data, commonly used in geographic information systems (GIS) to support range and intersection queries.

GiST: A generalized search tree framework unique to PostgreSQL, supporting customizable index types. It is suitable for complex data structures such as text, geography, and graphs.

The **B-Tree family** is efficient for point queries and has moderate update costs, making it suitable for most **OLTP** scenarios. R-Trees and GiST are more suitable for specialized scenarios, such as image processing, spatial data, and fuzzy matching. Different tree structures offer various trade-offs between storage complexity, query support, and maintenance costs. Database systems choose the appropriate indexing tree based on field types and query patterns.

5. What is a columnar database? Why is ClickHouse suitable for log analysis? What are the differences between ClickHouse and other databases? What are its features? What is MergeTree?

A **columnar database** stores data by **columns** rather than rows, which is beneficial for compression and skipping irrelevant data, especially suitable for large-scale aggregation, filtering, and grouping **OLAP** queries. **ClickHouse** is a typical high-performance columnar database designed for log analysis, real-time metrics platforms, and data warehouses. It can aggregate billions of rows in milliseconds and offers high compression, high-concurrency read/write capabilities, and excellent scalability.

Compared to traditional row-based databases like MySQL and PostgreSQL, **ClickHouse** excels in scenarios with more reads than writes, wide tables, and big data requiring fast **GROUP BY** and **WHERE** filters. However, it is not well-suited for frequent write-heavy workloads and transactional operations.

The core engine of ClickHouse is **MergeTree**, which divides data into multiple partitioned segment files (data parts), stores them in a sorted order by the specified primary key, and optimizes query performance through background asynchronous merging. MergeTree also supports partition pruning, sparse indexing, and vectorized execution. Various variants of MergeTree, such as ReplacingMergeTree, SummingMergeTree, and AggregatingMergeTree, are used for deduplication, aggregation, and metrics accumulation, making it ideal for log analysis, monitoring, and behavioral analytics.

6. Why not use Redis or DynamoDB for read-heavy and write-light scenarios? OLTP vs OLAP?

While Redis and DynamoDB also provide high-performance reads, they are fundamentally designed for OLTP (Online Transaction Processing) or caching/key-value storage, not complex large-scale aggregation and analysis. ClickHouse, however, is a columnar database specifically designed for OLAP (Online Analytical Processing) scenarios.

Redis is an in-memory database optimized for millisecond-level key-value read/writes but lacks support for complex SQL, multi-dimensional aggregation, joins, GROUP BY, and other analytical capabilities, making it inefficient for big data analysis. DynamoDB, Amazon's distributed NoSQL database, emphasizes high availability and elastic scaling but lacks native SQL support, complex analytical functions, and range aggregation capabilities. As a result, it is not ideal for large table scans.

ClickHouse, with its columnar storage, vectorized execution, sparse indexing, and MergeTree partition pruning, is designed specifically for structured analytics on petabyte-scale data, offering superior multi-dimensional analysis and scan performance. Therefore, in scenarios like log analysis, monitoring reports, and data visualization, ClickHouse excels at SQL analytics tasks that Redis and DynamoDB cannot handle, positioning it as a fundamentally different type of database.

OLTP (Online Transaction Processing) is designed for **high-concurrency, low-latency, small-batch read/write operations**, typical in order systems, inventories, user logins, and transactional business models. It emphasizes transactional consistency and real-time processing.

OLAP (Online Analytical Processing), on the other hand, is geared towards **large-scale data analysis tasks**, such as reporting systems, behavioral analysis, and log monitoring. It emphasizes large-scale aggregation, range queries, time-series analysis, and flexible query dimensions, focusing on read performance and query flexibility.

Design Patterns

Design Patterns, Factory Pattern, Singleton Pattern, Adapter Pattern, Decorator Pattern, Observer Pattern, Strategy Pattern, State Pattern

Design patterns are reusable, scalable, and maintainable best practice templates that have been identified in software development to solve common problems. They are not code frameworks but rather abstract design thinking and organizational structures that address "recurrent design issues in specific contexts." Design patterns typically describe the relationships, responsibilities, and collaboration between classes and objects, helping developers improve system flexibility, reduce coupling, and enhance readability. Common design patterns are categorized into three major types: **Creational** (e.g., **Factory Pattern, Singleton Pattern**), **Structural** (e.g., **Adapter, Decorator**), and **Behavioral** (e.g., **Observer, Strategy, State Patterns**). Mastering design patterns helps you write more elegant, scalable code and is a key indicator of strong system design capabilities.

The **Factory Pattern** is a creational design pattern that encapsulates the object creation process in a factory class, **hiding the details of object initialization**. It allows the creation of different objects based on input types. The client simply calls a unified interface provided by the factory without needing to worry about the specific creation logic. This pattern is often used when **object types are variable**, the creation process is complex, or depends on specific configurations, such as in **logging systems** with different Logger levels or database connection management. It increases the decoupling of the code and makes the system easier to extend.

The **Singleton Pattern** ensures that a class has **only one instance** throughout the entire system and provides a **global access** point. It is commonly used for managing global state, configuration, caches, thread pools, etc. The key implementation is to **privatize the constructor and return the unique instance via a static method**, often combining lazy or eager initialization with thread safety strategies. It reduces resource usage and avoids state conflicts, but misuse can lead to global dependencies and poor testability.

The **Adapter Pattern** is a **structural design pattern** used to **convert an existing interface into the interface expected by the client**, enabling otherwise incompatible classes to work together. The adapter typically wraps an existing object, calls methods of the original object internally, and converts the input/output format to the desired form. It is widely used in **integrating third-party libraries**, adapting legacy interfaces, and device drivers, providing compatibility without modifying the existing code.

The **Decorator Pattern** allows dynamic addition of functionality to **an object without modifying its structure**, offering a flexible alternative to inheritance. The core idea is to use multiple wrapper classes to enhance the functionality of the original object. For example, the BufferedInputStream decorator wraps FileInputStream to add buffering capabilities. The decorator pattern is commonly used in **UI rendering, logging chains, and permission control modules**, enabling high cohesion and low coupling in functional modules.

The **Observer Pattern** defines a one-to-many dependency, where a change in the state of the observed object **automatically notifies all dependent observers**, which then update themselves accordingly. This pattern is widely used in event systems, **GUI programming, and message subscription scenarios**. For example, **button clicks** trigger multiple callback functions, or **Redis key expiration** events notify listeners. It improves system responsiveness and scalability and is a classic implementation of the publisher-subscriber model.

The **Strategy Pattern** encapsulates **a set of algorithms into independent strategy classes**, allowing them to be interchangeably used and dynamically changing an object's behavior without modifying its core structure. The pattern defines a uniform operation interface, while specific strategy classes implement different logic. The client chooses the appropriate strategy based on the context. It is commonly used in **sorting, compression, routing, and authentication methods switching**. Compared to if-else or switch, the strategy pattern better adheres to the open-closed principle and is easier to extend and maintain.

The **State Pattern** allows an object to change its behavior when its internal state changes, as if it had changed its class. The core idea is to abstract the state logic into independent state classes, which can switch between states while holding a reference to the context. It is often used in **state machines, order processes, task scheduling, and TCP connections**. It eliminates the need for excessive conditional branching, makes state behaviors clear, easy to maintain, and follows the Single Responsibility Principle (SRP).

Distributed Systems

1. Basic Principles and Architecture Design of Distributed Systems

A **distributed system** consists of multiple autonomous computing nodes that communicate via a network to collaborate. Each node can operate independently, undergo failures or updates, while the system as a whole continues to function. This architecture offers advantages such as **high availability, horizontal scalability, and geographic fault tolerance**, and is widely used in internet services, big data platforms, and enterprise applications. However, it introduces typical challenges such as unreliable network communication, failure propagation between services, data consistency maintenance, clock synchronization issues, and difficulties in debugging and monitoring. Therefore, building a robust distributed system requires not only understanding business needs but also mastering collaboration between systems, fault tolerance, and recovery mechanisms.

The **CAP Theorem** (Consistency, Availability, Partition Tolerance) states that in a distributed system with network partitions, it is impossible to simultaneously achieve strong consistency and high availability. Design decisions need to be made between Consistency (C), Availability (A), and Partition Tolerance (P). For example, **ZooKeeper** leans toward CP (guaranteeing consistency while sacrificing some availability), whereas **Cassandra** leans toward AP (guaranteeing availability while relaxing consistency). To better align with engineering practice, the **BASE** theory introduces the concepts of "Basically Available," "Soft state," and "Eventual consistency," emphasizing system stability and scalability at the cost of temporary inconsistency. BASE is the foundation for designing high-availability systems in the internet domain.

Maintaining data consistency across replicas in distributed systems relies on consensus protocols. **Paxos** is the most complete theoretical algorithm, but it is complex to implement. **Raft**, a simplified version, consists of leader election, log replication, and state machine commitment stages and is more practical in real-world systems like Etcd and Consul. **ZooKeeper** uses the **ZAB protocol** for atomic broadcast and sequential consistency. Understanding these protocols helps in designing master-slave replication, leader election mechanisms, and distributed locks.

In **microservices architectures**, the number and status of service instances change dynamically, so relying on hardcoded IP addresses is not feasible. This requires a **Service Registry** (such as ZooKeeper, Etcd, or Consul) to record the availability and status of service nodes. Clients use polling or watching mechanisms to obtain real-time service lists and route requests to healthy instances based on load balancing policies. The service registry also handles health checks, service down notifications, and cluster failure recovery, playing a vital role in ensuring system autonomy and stability.

Caching is used to improve the performance of hot data access and reduce the load on databases. In distributed environments, **Redis Cluster** offers sharding, high availability, and master-slave replication, but it introduces consistency challenges. Common caching strategies include **Cache Aside** (application controls cache loading and invalidation), **Write-through** (write operations update both cache and database), and **Write-back** (first write to cache, then asynchronously flush to the disk). Additionally, mechanisms like **Bloom filters**, **local hot caches**, and **rate limiters** are used to mitigate issues like cache penetration, cache breakdown, and cache avalanches.

In distributed systems, data can be categorized into "**hot**" and "**cold**" data based on access frequency. Hot data is stored in high-performance storage (e.g., Redis, ClickHouse), while cold data is stored in low-cost storage (e.g., HDFS, S3) to reduce resource wastage. Business partitioning involves distributing users or data across different partitions/databases/servers (e.g., based on user_id, location, time) to enhance concurrency and system isolation, thereby preventing bottlenecks in a single point. This is essential for supporting millions of transactions per second (TPS).

Distributed file systems like **HDFS**, **Ceph**, and **GlusterFS** are used in big data platforms, where large files are sliced and stored across multiple nodes with replication mechanisms to ensure fault tolerance. **Object storage** solutions like **Amazon S3** and **MinIO** provide HTTP-based key-value storage interfaces and natively support multi-tenancy and version management, making them suitable for unstructured data, log archiving, and image storage.

Distributed databases like **TiDB** and **CockroachDB** support global transactions, using two-phase commit (2PC) or three-phase commit (3PC) to maintain cross-node consistency. **TCC** (Try-Confirm-Cancel) ensures eventual consistency through business logic compensation, commonly used in e-commerce and financial applications. **Outbox Pattern** in message transactions stores events and database writes in the same transaction, then asynchronously delivers them, often used in high-availability, asynchronous decoupled architectures. Distributed transactions are complex and require careful balancing of consistency, availability, and performance.

In Kafka, **partitions** enable horizontal scaling, **ISR** (In-Sync Replicas) ensures data redundancy, and **ACKs**, **offsets**, and **idempotent Producers** provide **At Least Once** or **Exactly Once** guarantees. **RabbitMQ** offers stronger transactional controls and plugin extension capabilities, but with slightly lower throughput. Key challenges in distributed messaging systems include **ordering guarantees**, **idempotent consumption**, **duplicate delivery handling**, and **backlog monitoring**.

High Availability (HA) design needs to cover all layers, including service registration, routing, caching, messaging, databases, and storage. Common methods include **master-slave replication** + **hot standby disaster recovery**, **failover**, **read-write separation**, **active-active centers**, and **multi-region active architecture**. By combining health checks, automatic circuit breakers, rate-limiting, fallback mechanisms, and redundant deployment, a system can maintain service availability even in the event of node failures.

A logging system should cover data collection (e.g., Filebeat, Fluentd), transmission (e.g., Kafka), cleaning and processing (e.g., Logstash, Flink), storage (e.g., Elasticsearch, ClickHouse), and query display (e.g., Kibana, Grafana). **Distributed tracing** systems such as **Jaeger** and **Zipkin** use TraceID and SpanID to inject context throughout the lifecycle of requests, enabling the reconstruction of microservice call chains and pinpointing performance bottlenecks. **OpenTelemetry** provides a unified data model and collection interface, enhancing cross-service observability.

The gateway layer serves as the unified entry point for a system, responsible for authentication, rate-limiting, protocol conversion, canary releases, traffic monitoring, etc. **Nginx** supports various algorithms like IP hash, least connections, and weighted round-robin. **Envoy** supports gRPC, dynamic routing, and advanced plugin extensions. **Kong** provides API management and plugin-based security control. After traffic enters, **Kubernetes services**, **Ingress**, and **HPA (Horizontal Pod Autoscaler)** automatically adjust the number of backend replicas, enabling elastic load balancing. Kafka's partition mechanism also inherently supports parallel consumption. The overall architecture should combine "gateway entry + load balancing + async queues + microservice decoupling + data caching" to achieve high scalability, elasticity, and fault tolerance.

2. Practical Components of Distributed System Architecture

In modern backend system design, building high-availability, high-performance, and scalable distributed services relies on a mature ecosystem of system components. Compared to traditional object-oriented and operating system-level programming, distributed architecture places greater emphasis on **engineering system thinking** and

resilience assurance. This requires developers to possess deep understanding and hands-on experience with various middleware, infrastructure tools, and service governance platforms.

This chapter focuses on core components frequently covered in technical interviews, including **Redis** (caching and distributed locking), **Kafka** (message middleware), **Nginx** (high-performance proxy), **Kubernetes** (container orchestration and elastic scheduling), and **API Gateway** (microservices gateway and traffic governance). It covers their working principles, application scenarios, key configurations, failure recovery strategies, and common interview questions. Through systematic organization, this section helps candidates develop modular, automated, and elastic service governance capabilities in real-world projects, while also showcasing their engineering maturity in designing high-availability architectures, ensuring service reliability, and troubleshooting distributed systems.

2.1 Redis

Keyword

Redis is an open-source, high-performance in-memory key-value store that supports rich data structures such as strings, hashes, lists, sets, and sorted sets. It is built on a single-threaded event loop model and achieves extremely fast read/write performance through efficient memory operations and I/O multiplexing (e.g., epoll). Redis is widely used for caching, distributed locking, message queues, leaderboards, and session management. It offers persistence mechanisms (RDB snapshots and AOF append-only logs) to ensure data safety, supports high availability through replication and Sentinel, and enables horizontal scalability via Redis Cluster.

RDB (Snapshot Persistence) is one of Redis's persistence options, which periodically saves the in-memory dataset to disk as a snapshot (RDB file). This method provides fast recovery and compact file size, making it suitable for cold backups and disaster recovery. However, because it saves data at intervals, some recent updates may be lost if Redis crashes. Snapshots can be triggered manually using save or bgsave, or configured with automated rules like save 900 1. RDB is ideal in development environments or in scenarios where persistence is not critical but fast recovery is desirable.

AOF (Append-Only File Persistence) is another Redis persistence mechanism that logs every write command to an append-only file. AOF provides a higher level of data safety compared to RDB, as it captures all write operations and supports recovery even after unexpected crashes. AOF supports three write policies: always sync (safest but slowest), sync every second (default), and OS-managed sync (fastest but risk of data loss). Redis periodically rewrites the AOF file to reduce file size. AOF is suitable for systems with strong data integrity requirements, such as financial, e-commerce, and log platforms.

TTL (Time To Live) is Redis's key expiration mechanism, allowing a time limit (in seconds or milliseconds) to be set for a key, after which it is automatically deleted. Expiration is implemented via lazy deletion (checked upon access) and periodic deletion (random sampling by background processes). Commands like expire, ttl, and pttl help manage TTLs. TTL is widely used in caching systems to prevent memory bloat and release space, and is common in session control, verification code expiration, and rate limiting.

LRU (Least Recently Used) is a data eviction policy used by Redis when memory reaches the configured maximum (maxmemory). LRU prioritizes the removal of the least recently accessed keys. Redis implements an approximate LRU algorithm using sample-based eviction. Other policies include LFU (Least Frequently Used), TTL-based eviction, and random eviction. Properly configuring eviction strategies is essential in high-concurrency scenarios to prevent cache avalanche and ensure service availability.

Sentinel is Redis's high-availability deployment solution. It monitors master node status and performs automatic failover by promoting a replica to master if the current master becomes unavailable. Sentinel also notifies clients to update master addresses, ensuring uninterrupted service. Sentinel runs as a separate process and requires deployment of multiple instances for quorum-based decision-making. It's widely used in medium to large-scale production systems like financial services, order platforms, and user session caches.

Redis Cluster is the official distributed mode of Redis, suitable for scaling beyond a single machine's memory. Redis Cluster hashes keys into 16,384 slots and distributes these slots across multiple nodes, achieving automatic data sharding. Each master node handles a subset of slots and can have replicas for redundancy. Redis Cluster supports read/write separation, automatic failover, and slot rebalancing, but does not support multi-key operations across slots (unless using hash tags). It is ideal for high-concurrency applications such as real-time leaderboards, social graphs, and shopping carts.

Pipeline is a Redis performance optimization technique that allows clients to send multiple commands in a single network round trip, reducing latency and increasing throughput. Note that Pipeline does not guarantee atomicity; it is purely a batching mechanism. It's commonly used in scenarios like bulk data loading, cache warm-up, and scheduled cleanup tasks.

Lua Script (EVAL) is Redis's server-side scripting mechanism that enables atomic execution of multiple commands in one transaction. Developers can write business logic in Lua and execute it via the EVAL command to ensure atomicity. Lua scripts are widely used for safe distributed lock release (check before delete), batch operations, and access control. Proper use of Lua scripts simplifies client logic and enhances consistency in high-concurrency environments.

Big Key refers to a single Redis key that consumes a large amount of memory or contains a massive number of elements (e.g., Lists, Hashes, Sets, Sorted Sets with hundreds of thousands of entries). Since Redis is single-threaded, operations on Big Keys can cause blocking and degrade overall performance, leading to latency spikes or timeouts. Developers should avoid creating Big Keys or split them into smaller keys. Use redis-cli --bigkeys to identify potential Big Keys. Monitoring Big Keys is crucial in scenarios like user follow lists, messaging systems, or large product catalogs.

Distributed Lock (SET NX EX / RedLock) is a concurrency control pattern commonly implemented using Redis to coordinate access to shared resources across multiple processes. The core command is SET key value NX EX ttl, which sets the key only if it doesn't exist and sets an expiration time. To prevent unsafe lock release, Lua scripts are used to validate before deletion. RedLock is a multi-node locking algorithm proposed by Redis's creator for higher reliability. Common use cases include inventory deductions, ensuring order uniqueness, and enforcing idempotency.

Cache Penetration / Breakdown / Avalanche are common failure modes in Redis caching systems under high concurrency. **Cache Penetration**: Requests for non-existent keys bypass the cache and hit the database each time. Solutions include Bloom filters and caching null results. **Cache Breakdown**: A hot key expires and receives massive concurrent requests, all hitting the backend. Solutions include mutex locks and proactive refresh. **Cache Avalanche**: Many keys expire at the same time, overloading the database. Solutions include setting randomized expiration times and rate limiting. These issues are critical to address in high-traffic scenarios like flash sales, user login, and promotional events.

Pub/Sub (Publish/Subscribe) is Redis’s real-time messaging model that allows clients to subscribe to channels and receive messages as they are published. Pub/Sub supports fast, event-driven communication, making it suitable for broadcasting, chat systems, and configuration updates. However, it lacks persistence and delivery guarantees, so it’s not suitable for critical messaging requirements.

Slowlog (Slow Query Log) is Redis’s built-in diagnostic tool for recording commands that exceed a specified execution time threshold. Use SLOWLOG GET to view recent entries and configure the threshold with slowlog-log-slower-than (default is 10,000 microseconds). Slowlog only captures command latency, not the reasons for slowness (e.g., large keys or blocking commands), and is useful for performance troubleshooting.

BitMap / HyperLogLog / GEO are three specialized Redis data structures designed for specific modeling and query needs. **BitMap**: Efficiently stores and manipulates boolean states such as user check-ins or activity flags; supports bit counting and bitwise operations; ideal for large-scale user tracking. **HyperLogLog**: A probabilistic data structure for estimating the cardinality of unique elements using fixed memory (~12KB); ideal for UV and unique device count metrics. **GEO**: Stores and queries geospatial data like longitude and latitude, supporting distance calculation, range queries, and sorting by proximity; suitable for "nearby people/stores" use cases. These structures provide Redis with flexible, high-performance modeling capabilities commonly used in marketing platforms, user profiling, and location-based services (LBS).

2.1.1 What is cache breakdown? How does it occur in real systems, and how can it be mitigated from an engineering perspective?

Cache breakdown refers to the scenario where **a specific hot key expires under high-concurrency conditions, causing all requests to bypass the cache and directly hit the database or backend service**. This sudden spike can overwhelm the database, leading to service outages. This typically happens when a frequently accessed key expires just as many concurrent requests are arriving. Common mitigation strategies include: (1) using mutual exclusion locks (e.g., SETNX) to ensure only one request rebuilds the cache while others wait or fail quickly; (2) proactive cache refresh to avoid expiration; (3) fallback local caching (e.g., Caffeine); (4) using request deduplication or message queues to throttle traffic. Cache breakdown is a classic interview question that tests one’s understanding of designing a pressure-resistant cache-database system.

2.1.2 What is the difference between cache avalanche and cache breakdown? How do you handle large-scale cache expiration?

Cache avalanche refers to the situation where many cached keys expire in a short period, causing a large number of requests to bypass the cache and overwhelm the database. Unlike cache breakdown (which targets a single hot key), avalanche affects the entire cache layer. Causes include setting uniform TTLs during pre-warming, pre-warm task failures, or abrupt cache system restarts. Mitigation strategies: (1) introduce randomness to TTLs to avoid synchronized expirations; (2) proactively refresh important caches; (3) build multi-level caches (e.g., Redis + local); (4) use rate limiting and fallback mechanisms (e.g., Sentinel, Hystrix); (5) deploy replicas or hot-standby Sentinel nodes. Avalanche prevention focuses on "temporal distribution" of expiration and requires systematic design thinking and traffic pattern awareness.

2.1.3 What is cache penetration? Is it always an attack? How can you defend against it?

Cache penetration occurs when clients repeatedly request keys that are nonexistent in both the cache and database. This leads to repeated database access and bypasses the cache protection. It may be caused by malicious attacks (random keys), user errors, or bugs. Defense strategies include: (1) cache null values for non-existent data with short TTLs; (2) use Bloom Filters or Cuckoo Filters to intercept definitely non-existent keys; (3) apply blacklists, rate limiting, or CAPTCHA for abuse prevention. It tests the system's robustness and ability to guard against invalid access patterns.

2.1.4 Does Redis's transaction mechanism (MULTI/EXEC) provide full database transaction semantics? What are its pitfalls?

Redis supports MULTI/EXEC for transactional operations, but it differs from relational DB transactions. It guarantees command ordering and atomic execution but does not support rollback or isolation levels. All commands are queued after MULTI and executed in batch after EXEC. Key caveats: (1) syntax errors are caught during queuing, not at execution; (2) even if some commands fail, others will still execute — no rollback; (3) use WATCH for optimistic locking. Redis transactions are lightweight and suitable for predictable batch updates, not for strict consistency guarantees.

2.1.5 What eviction strategies does Redis support? Which scenarios are they suited for?

When Redis hits its maxmemory limit, it evicts keys based on the configured maxmemory-policy. Common policies: noeviction (default, write fails), allkeys-lru (evict least recently used), allkeys-random, volatile-lru (LRU among keys with TTL), volatile-ttl (evict soon-to-expire keys), volatile-random. Use cases: allkeys-lru for general caching, volatile-lru for temporary data, noeviction for critical key protection. Redis uses an approximate LRU algorithm (sample-based), not strict LRU.

2.1.6 How does Redis handle key expiration? Why does it use a combination of strategies?

Expired keys are not deleted immediately. Redis combines: (1) lazy deletion (checked upon key access); (2) periodic deletion (background thread checks random keys, by default 10 times/sec with 20 keys per run); (3) memory-pressure-triggered eviction (expired keys prioritized). This design balances performance and real-time needs. Redis favors a "lazy + controllable" cleanup approach to avoid spikes during high-concurrency.

2.1.7 Is Redis single-threaded? Why can it support high concurrency?

Yes, Redis runs its main thread in a single-threaded model (aside from persistence and AOF rewrite). It uses epoll + I/O multiplexing to handle connections and commands efficiently. High performance comes from: (1) all operations are in-memory; (2) optimized core data structures (skip lists, hash tables); (3) minimal context switching and locking; (4) lightweight RESP protocol. For scaling, use multi-instance deployment, Redis Cluster, and pipelining. Redis performance is rooted in simplicity and focus, not multi-threading.

2.1.8 What persistence mechanisms does Redis offer? How do RDB and AOF differ?

Redis provides RDB (snapshot) and AOF (append-only log). RDB saves the dataset at intervals as .rdb files — suitable for backups and fast recovery but may lose recent data. AOF logs every write operation and supports always, everysec (recommended), and no fsync policies. AOF offers better data durability but has larger file sizes and slower recovery. Redis 4.0 introduced hybrid persistence combining RDB recovery speed with AOF reliability. In interviews, discuss trade-offs: e.g., disable persistence for caching, use AOF for critical data, evaluate fork latency, and COW impact.

2.1.9 What is Redis distributed locking? Can SET NX alone provide reliable locking? Is RedLock recommended?

Distributed locks using Redis involve setting a key only if it doesn't exist (SET key value NX EX ttl). This ensures mutual exclusion with automatic expiry. Challenges include: (1) locks expiring before task completion; (2) accidental deletion of someone else’s lock. Solutions: include a unique ID and use Lua scripts to validate ownership before deletion. RedLock, proposed by Redis's author, coordinates locks across multiple instances. It improves fault tolerance but is complex and debated in edge cases like clock drift or partitions.

2.1.10 How to implement a leaderboard using Redis? How does Sorted Set work?

Sorted Set (ZSet) is ideal for leaderboard use cases (e.g., scores, rankings). It combines a skip list and a hash table, allowing each member to have a score and automatic ordering. Use ZADD to insert, ZREVRANGE to get top N, ZINCRBY to increment scores, and ZREMRANGEBYRANK to prune. Sorted Sets support efficient top-N queries, range lookups, and real-time updates. They're commonly used in gaming, e-commerce, and user engagement ranking systems.

2.1.11 How to implement a delayed queue with Redis? How is it different from a regular message queue?

While Redis is not designed specifically as a message queue, it can implement delayed queues effectively using the Sorted Set data structure by combining the score field with a timestamp. The idea is to insert tasks into a Sorted Set where the score is the execution time. A polling thread regularly checks for tasks whose timestamps are less than or equal to the current time and processes them. This design, often referred to as a time-wheel delay queue, is widely used in scenarios like delayed SMS, order timeout processing, and scheduled jobs. The commands involved include ZADD (add task), ZREVRANGEBYSCORE (query expired tasks), and ZREM (remove processed tasks), often wrapped in Lua scripts to ensure atomic and idempotent consumption. Compared to

Kafka or RabbitMQ, Redis delayed queues are more suitable for lightweight workloads with moderate volume and lower reliability requirements.

2.1.12 How does Redis implement globally unique IDs? What are the available approaches?

Generating globally unique IDs is a common requirement in distributed systems. Redis offers simple and high-performance solutions such as using the INCR or INCRBY commands, which return incremented integers—naturally unique and ordered. This approach is suitable for order IDs and sequence numbers, capable of handling millions of requests per second. However, concentrating all requests on a single key can create a bottleneck. A more scalable design combines timestamps, business codes, and Redis-generated ID segments. Additionally, clients can cache a range of IDs in memory to reduce Redis calls. For more complex systems, the Snowflake algorithm is often used, and Redis can help assign unique worker IDs to ensure cross-datacenter uniqueness.

2.1.13 Is Redis suitable for message queuing? How does it differ from Kafka?

Redis can serve as a lightweight message queue using List (via LPUSH + BRPOP) or Stream (via XADD + XREADGROUP). Lists are simple and support blocking consumption, suitable for task queues and logging. Streams, introduced in Redis 5.0, support consumer groups, acknowledgment, and offset tracking, resembling Kafka's model. Redis queues are easy to use with low latency but lack persistence, message replay, partitioning, and backlog management found in Kafka. Thus, Redis is ideal for short-lived tasks, while Kafka is better for decoupled systems, asynchronous transactions, and analytics pipelines.

2.1.14 What is the sharding mechanism behind Redis Cluster? How does it achieve horizontal scalability?

Redis Cluster is Redis's native distributed solution that partitions keys across 16,384 hash slots. These slots are evenly distributed among cluster nodes, with each node handling a subset of the keys. Clients connect to any node to retrieve the slot mapping table and then communicate directly with the appropriate node—there is no central proxy. When scaling, administrators use resharding to move slots between nodes dynamically. Redis Cluster also supports high availability through master-replica structures. Limitations include the lack of multi-key transaction support across slots (unless using hash tags), and the need to handle MOVED redirects correctly.

2.1.15 How to troubleshoot Redis slow queries? What tools and best practices can improve performance?

Redis defines slow queries as commands exceeding a certain execution time threshold (default 100ms). These are often caused by large keys, blocking operations, or inefficient commands. Use the built-in SLOWLOG tool (SLOWLOG GET) to inspect recent slow commands. Supplement this with MONITOR or INFO commandstats for real-time or aggregate insights. Best practices: avoid full-keyspace scans (e.g., KEYS), control the size of sets and hashes, and clean up oversized keys promptly. Batch commands using pipeline to reduce network round-trips. Configure timeout to prevent hanging clients and use maxmemory-policy alongside TTLs to ensure timely memory reclamation.

2.2 Kafka

Keyword

Kafka is a high-throughput, persistent, distributed streaming message platform based on a publish-subscribe model. It is widely used in scenarios such as log collection, distributed tracing, stream processing, and asynchronous decoupling. Kafka organizes messages by **Topic**, which is subdivided into **Partitions** for parallel consumption and ordering guarantees. It follows a pull model, where **Consumers** manage their own offset progress. Kafka achieves high write efficiency through sequential disk writes and zero-copy technology, and ensures data availability through the **ISR (In-Sync Replicas)** mechanism. Kafka supports **Producer** idempotency, **transaction semantics**, and automatic **Consumer Group** rebalancing.

Producer is the client component responsible for **writing messages to Kafka Topics**. It supports async/sync send, batching, message compression, idempotent writes (to avoid duplication), and transactional guarantees (to ensure multi-write consistency). Messages are routed to specific partitions based on the message key, enabling ordered delivery within a partition. Internally, it uses memory buffers, a sending thread, and retry logic to enhance throughput and reliability. A common use case is when an order service asynchronously publishes order data to a Kafka Topic, which is then consumed by inventory, risk control, and recommendation systems.

Consumer is the client that subscribes to Kafka and pulls messages. It manages the offset (consumption progress), which can be auto-committed (less safe) or manually committed (more controlled). Kafka consumers use a pull model and periodically poll for batches of messages. Features include batch processing, deserialization, and idempotent consumption. When used with Consumer Groups, partitions are automatically balanced across consumers for parallelism. Common use cases include log analytics, recommendation engines, and stream processing platforms like Flink and Spark.

Topic is the logical classification unit in Kafka. Producers publish to Topics and consumers subscribe to them. Topics are typically organized by business logic (e.g., order-topic, payment-topic, user-behavior-topic) and can be configured with retention time, compression policy, etc. A Topic is split into **Partitions** for scalability.

Partition is Kafka’s fundamental unit of parallelism. Each Topic consists of multiple Partitions, each maintaining an ordered log stored on disk. Messages are ordered within a partition but not across partitions. A partition can only be consumed by one consumer within a Consumer Group, ensuring mutual exclusivity. For example, a high-throughput e-commerce order stream might be split across 10–20 partitions for parallel processing.

Offset is the position of the next message to be read in a partition. Kafka retains all messages for a configured time or size and does not delete consumed messages. Offsets enable replay, recovery, and duplicate handling. Offsets can be auto or manually committed; manual commit is recommended for critical systems like payments.

Consumer Group enables Kafka’s parallel consumption model. Each group has multiple consumers, and Kafka distributes partitions among them such that each partition is consumed by only one group member. Groups operate independently and can implement broadcast models. For example, one group can handle real-time recommendations, while another performs data aggregation.

Broker is a Kafka server that handles client requests, stores messages, manages replicas, and handles reads/writes. A Kafka cluster consists of multiple Brokers that share the workload. Producers send messages to the appropriate partition’s broker; consumers pull from them. A typical cluster starts with 3–5 Brokers for high availability and throughput.

Controller is a special Kafka Broker responsible for managing cluster control tasks such as topic creation, partition leadership elections, and broker status. Only one Controller is active at a time; if it fails, a new one is elected. Controller health is vital for cluster stability.

ISR (In-Sync Replicas) are the set of replicas that are up-to-date with the partition leader. Kafka’s reliable write depends on ISR. Only when a message is acknowledged by a quorum of ISR replicas is it considered successfully written. Using acks=all ensures maximum durability. ISR is critical in financial, e-commerce, and risk-sensitive systems.

Retention defines how long messages are retained in Kafka, even after being consumed. Messages are kept for a configured duration (e.g., 7 days) or until disk size limits are reached. Useful for reprocessing, recovery, and batch analytics.

Log Compaction is a retention strategy that keeps only the latest message for each key, removing older versions. Ideal for state synchronization or configuration updates (e.g., config centers).

Idempotence enables Kafka Producers to avoid message duplication. When enabled (enable.idempotence=true), Kafka deduplicates messages by tracking producer IDs and sequence numbers. This is critical in payment, invoice, and transaction systems.

Transaction in Kafka allows producers to atomically write to multiple partitions/topics. Producers use beginTransaction, commitTransaction, and abortTransaction. Consumers using read_committed mode avoid reading uncommitted data. Transactions ensure exactly-once semantics, suitable for operations like order + billing updates.

Acks define the producer's message durability guarantee. Options: acks=0 (no confirmation), acks=1 (leader confirmed), acks=all (all ISR replicas confirmed). Higher acks = stronger durability but more latency.

Rebalance automatically redistributes partitions when group membership or partition count changes. It causes short pauses. Cooperative rebalancing reduces impact. Systems like real-time risk control should optimize task handoff during rebalances.

Zookeeper / KRaft are metadata management layers. Zookeeper (legacy) handles metadata and leader election. KRaft (Kafka Raft, 2.8+) eliminates the Zookeeper dependency, simplifying management and boosting performance.

Poll / Fetch are the consumer mechanisms for pulling data. poll is called periodically to fetch messages. Batch sizes are controlled by fetch.min.bytes, max.poll.records, etc., and should be tuned based on workload.

Log Segment is the physical file structure of a Kafka partition. Messages are sequentially written to segments. When a segment reaches a threshold, a new file is created. Old segments are deleted or compacted.

Leader / Follower are replica roles in a partition. The Leader handles all reads and writes; Followers replicate asynchronously. If the Leader fails, a new one is elected from the ISR. This ensures consistency and availability.

Replica is a copy of a partition. Kafka supports multiple replicas (e.g., 3) across Brokers. One is the Leader; others are Followers. This design ensures data durability and availability even during failures.

Rack Awareness allows Kafka to distribute replicas across different racks or data centers via broker.rack configuration. It enhances fault tolerance and disaster recovery in multi-datacenter deployments.

Kafka Connect is Kafka's official integration framework for moving data between Kafka and external systems (e.g., DBs, logs, cloud storage). It supports plugins, auto-scaling, fault tolerance, and offset tracking. Common in ETL pipelines, data lakes, and CDC ingestion.

Kafka Streams is a lightweight Java library for stream processing directly within Kafka clients. It supports windows, joins, aggregations, state management, and exactly-once semantics. Ideal for low-latency, in-memory processing in real-time systems (e.g., product popularity tracking).

Exactly-once Semantics (EOS) guarantees that messages are processed exactly once end-to-end. Kafka combines idempotent producer, transactional write, and atomic offset commit to achieve EOS. It’s essential for highly accurate systems like payments and financial reconciliations.

Watermark is a concept from stream processing frameworks (e.g., Flink, Beam) to represent event time progress and trigger windowed computations. Kafka doesn’t natively support Watermark, but it is crucial when integrated with Flink for use cases like “remind if order unpaid for 5 minutes.”

Dead Letter Queue (DLQ) is a Kafka mechanism for handling failed messages. Messages that cannot be processed due to errors or retries can be sent to a DLQ for later analysis, compensation, or manual review. Useful in scenarios like user registration or legacy data replay.

Throttling limits producer/consumer throughput to protect system resources. Kafka supports quota-based throttling and replica lag-based throttling. It prevents cluster overload in peak or skewed traffic scenarios.

Lag measures consumer delay — the difference between the latest offset and the consumer’s offset. Large lag implies the consumer is falling behind. Monitoring lag helps manage scaling and SLA in systems like alerting or real-time analytics.

Compaction Topic enables a snapshot-like retention policy, preserving only the latest value per key. Unlike time-based retention, it keeps state information compact and current. Ideal for account balance or config centers.

Metadata in Kafka includes global cluster information like topic structure, partition assignment, broker status, and controller info. Clients fetch metadata at startup to route requests. Inconsistencies or delays in metadata can cause errors, so monitoring this is crucial in production.

2.2.1 How is offset managed in Kafka? What's the difference between auto and manual commit?

Kafka consumers track the position of consumed messages in each partition using an offset. Offsets can be committed automatically or manually. Auto-commit periodically stores offsets in Kafka's internal topic (__consumer_offsets), which is simple but may lead to duplicate or lost messages during failures. Manual commit provides more control—developers commit only after successful message processing, ensuring atomicity between business logic and offset tracking. Production environments typically prefer manual commit, optionally combined with async commit and transactions for stronger consistency.

2.2.2 How does Kafka ensure message ordering? Can strict order be maintained within a consumer group?

Kafka guarantees message order at the partition level. To ensure ordering, producers must send messages with the same key to a specific partition. Within a consumer group, each partition must be processed by only one consumer thread to preserve order. Global ordering across the entire topic requires using a single partition, which sacrifices concurrency. Kafka achieves a balance between ordering and scalability through partition-based parallelism.

2.2.3 How does Kafka's producer idempotency work? Can it guarantee no duplicate messages?

Introduced in Kafka 0.11, producer idempotency ensures no duplicate writes in the event of retries. Each producer is assigned a unique PID and sequence numbers. Even if the same message is sent multiple times due to network failures, the broker identifies and discards duplicates based on these identifiers. Enable this feature with enable.idempotence=true. While idempotency prevents duplication within a single partition, cross-partition or multi-topic writes require transactional APIs for full consistency.

2.2.4 What is Kafka's ISR (In-Sync Replica) mechanism and how does it ensure availability?

Kafka uses ISR (In-Sync Replicas) to maintain reliable replication. Each partition has one leader and multiple followers; only followers fully synced with the leader are part of the ISR. When producers write data, Kafka ensures the message is replicated to all ISR members before acknowledging success. If the leader crashes, Kafka elects a new leader from the ISR. While ISR improves fault tolerance, if too few replicas stay in sync, write throughput may suffer. Kafka may remove lagging replicas from the ISR to reduce pressure.

2.2.5 How does Kafka achieve high throughput? What optimizations distinguish it from traditional queues?

Kafka leverages sequential disk I/O and OS page cache to avoid random disk access. It uses batch sends, zero-copy transfer (sendfile), and compression (Snappy, etc.) to minimize network overhead. Unlike traditional message queues, Kafka is log-based, using append-only writes that are easy to persist and retrieve. Consumers can read at their own pace, enabling backpressure and flexible traffic control. Kafka's distributed partition model enables horizontal scalability and distinguishes it from systems like RabbitMQ or ActiveMQ.

2.2.6 How is message persistence handled in Kafka? When are messages deleted?

Kafka persists messages to disk as log segments, organized by partition and time. Persistence uses page cache and is governed by flush.policy. Messages are not deleted immediately after consumption. They are retained based on time (retention.ms) or size (retention.bytes) policies. Log cleaner threads asynchronously remove old data. With log compaction enabled, Kafka keeps only the latest message per key, which is useful for state-sync scenarios.

2.2.7 How does Kafka achieve exactly-once semantics? What parameters are required?

Kafka's exactly-once semantics rely on: (1) idempotent producers (enable.idempotence=true), (2) transactional APIs (transactional.id + begin/commitTransaction), and (3) committing offsets only after business logic is complete. Producers group multiple writes into atomic transactions. Consumers use read_committed mode to avoid reading uncommitted messages. This setup ensures a single message is processed exactly once throughout the pipeline.

2.2.8 How does Kafka's Rebalance mechanism work? What's the impact on consumers?

Rebalancing occurs when consumer group membership changes (e.g., a consumer joins or leaves). Kafka redistributes partitions among consumers, causing brief interruptions. Old consumers release partitions, and new ones resume from the last committed offset. Kafka 2.4+ introduced cooperative sticky rebalancing to minimize disruption. To optimize rebalancing, tune parameters like session.timeout.ms, heartbeat.interval.ms, and max.poll.interval.ms to balance fault tolerance and responsiveness.

2.2.9 How does Kafka handle consumer lag? Are there monitoring tools for backlogs?

Consumer lag happens when processing falls behind message production. Kafka exposes metrics like lag, consumer lag, bytes in/out, and request handler idle via JMX, Prometheus, or Confluent Control Center. Solutions include: increasing consumer count, using batch processing, alerting and throttling slow consumers, and adjusting max.poll.records and fetch.min.bytes for better control. Monitoring lag is crucial in real-time systems like fraud detection and alerting.

2.2.10 How does Kafka balance message ordering with consumption throughput?

Kafka partitions enable both ordering (within each partition) and parallelism (across partitions). To maintain order, producers must send related messages (e.g., by user ID or order number) to the same partition. Consumers must avoid concurrent processing of the same partition. Interviews often assess whether candidates can design for "local ordering + high throughput" by carefully choosing partitioning strategies, synchronized consumers, or order recovery mechanisms at the application layer.

2.3 Kubernetes

Keyword

Kubernetes (K8s) is an open-source container orchestration platform used to automate the deployment, scaling, load balancing, and self-healing of containerized applications. Its core components include the API Server, Scheduler, Controller Manager, Kubelet, Kube-Proxy, and etcd. Kubernetes uses a declarative resource model (YAML) combined with controllers (Deployment, StatefulSet) to manage container state. It supports service discovery, Ingress routing, ConfigMap/Secret injection, and horizontal auto-scaling (HPA). K8s features plugin-based support for networking (CNI), storage (CSI), and access control (RBAC), and supports extensibility through CRDs, making it widely adopted in microservice and cloud-native architectures.

Pod A Pod is the smallest deployable unit in Kubernetes, encapsulating one or more containers that share network, storage, and lifecycle. Containers in a Pod collaborate, sharing an IP address, hostname, and volumes. Pods are typically scheduled onto a Node by the Kubernetes Scheduler. In practice, a web app might run in a single-container Pod, while backup or logging tasks might use multi-container Pods for cooperative work.

Deployment A Deployment manages stateless applications, offering replica control, rolling updates, and rollback capabilities. Users define the desired number of Pods, container images, and ports, and the controller ensures alignment with the target state. It is the most common workload controller for managing web services, APIs, and microservices.

StatefulSet StatefulSet is used to manage stateful applications. Unlike Deployment, it supports ordered Pod creation, stable network identifiers, and persistent volume claims (PVCs) for each Pod. It’s suitable for databases (MySQL, PostgreSQL), Zookeeper, Kafka, and similar systems that require data consistency and persistent identity.

Service A Service provides a stable network endpoint for a group of Pods and implements load balancing and service discovery. Types include ClusterIP (internal only), NodePort (exposes service via node IP and port), LoadBalancer (cloud LB integration), and ExternalName (DNS redirection). Services decouple consumers from changing Pod IPs and are essential for service-to-service communication.

Ingress Ingress is a K8s resource that manages HTTP/HTTPS routing, allowing external access to internal services using domain and path-based rules. Combined with Ingress Controllers (e.g., Nginx, Traefik), it supports TLS termination, rewriting, canary releases, and unified gateways. Ingress is commonly used to build API gateways and manage public-facing access.

ConfigMap / Secret ConfigMap and Secret provide configuration injection into Pods. ConfigMap is used for non-sensitive data (e.g., configs, ENV variables), while Secret handles sensitive data (e.g., passwords, TLS certs). Both can be mounted as Volumes, environment variables, or command arguments, and support hot reload through volume-based mounting combined with file watchers or sidecar pattern.

Volume / PVC / PV Volumes persist data across container restarts. PersistentVolume (PV) and PersistentVolumeClaim (PVC) decouple storage provision from Pod requests. Users define PVCs, and the system binds them to PVs based on StorageClass definitions, enabling dynamic provisioning. This model is ideal for databases, caches, and user-upload scenarios requiring durable storage.

ReplicaSet / HPA ReplicaSet ensures the desired number of Pod replicas are maintained. Horizontal Pod Autoscaler (HPA) dynamically adjusts the number of replicas based on CPU, memory, or custom metrics. HPA is effective in high-traffic or burst scenarios like sales events, helping manage load and reduce infrastructure costs.

Taint / Toleration / NodeSelector / Affinity These features control Pod scheduling. Taints mark nodes as unschedulable unless tolerated. NodeSelector provides basic node targeting, while Affinity enables advanced Pod-to-Pod or Pod-to-Node placement rules. These tools are used for GPU task isolation, production/test separation, or high-availability strategies.

InitContainer / Sidecar InitContainers run before main containers and are used for setup tasks. Sidecars run alongside main containers to handle auxiliary tasks like logging, proxies, and config syncing. The sidecar pattern is prevalent in service meshes like Istio for traffic and security controls.

Pod Probe Probes check container health. livenessProbe triggers restarts if a container is unresponsive; readinessProbe removes Pods from load balancers if they aren't ready; startupProbe delays other probes until startup completes. Proper configuration improves system stability and fault isolation.

Namespace / RBAC Namespaces provide logical resource isolation. RBAC (Role-Based Access Control) restricts actions based on user roles and permissions. Together, they support multi-environment (dev/test/prod) deployments and multi-tenant access control. Roles and bindings are created per namespace or cluster-wide to assign permissions.

Core Components (Kubelet, Kube-Proxy, etc.)

- **Kubelet**: Node agent that manages Pod lifecycles.
- **Kube-Proxy**: Implements service networking and load balancing.
- **Scheduler**: Assigns Pods to Nodes based on constraints and availability.
- **Controller Manager**: Runs core control loops (e.g., Deployment, Job).
- **API Server**: Central entry point for all cluster operations.
- **etcd**: Key-value store for all cluster data. These components form the control and data planes of Kubernetes, enabling resilient, declarative container orchestration.

2.3.1 What's the difference between Deployment and StatefulSet in Kubernetes? In what scenarios should each be used?

Deployment and StatefulSet are both controllers in Kubernetes used to manage Pod replicas. Deployment is suitable for stateless services where Pods do not require persistent identity or storage. All replicas are identical and can be created, updated, or scaled freely, making it ideal for web services and microservices. StatefulSet, on the other hand, is used for stateful services where Pods need stable network identities (e.g., pod-0, pod-1), ordered deployment, and persistent volume binding. It is commonly used for databases, Kafka, Zookeeper, etc. In interviews, you can highlight scenarios like Redis Cluster or MySQL replication where StatefulSet ensures data consistency and sequence.

2.3.2 What are the phases of a Pod lifecycle, and how is the restart policy controlled?

Pod lifecycle phases include Pending, Running, Succeeded, Failed, and Unknown. Pending means the Pod is waiting for scheduling or resource allocation. Running means one or more containers are active. Succeeded and Failed reflect container termination states. The restartPolicy controls behavior after container exit: Always (default, for continuous workloads), OnFailure (only restarts on failure), and Never. Deployment uses Always, while Job may use OnFailure or Never. Additionally, Kubelet leverages probes (liveness/readiness) to determine restart needs. Containers within a Pod may have separate lifecycles, such as InitContainers and main containers.

2.3.3 What types of Services are available in Kubernetes, and what are their use cases?

Service types include ClusterIP (default, internal access), NodePort (accessible via node IP and port), LoadBalancer (exposes via cloud load balancer), and ExternalName (DNS alias to external service). ClusterIP is used for internal microservice communication. NodePort enables basic external access in self-managed clusters. LoadBalancer is suited for cloud deployments needing direct public access. ExternalName helps integrate external systems via DNS. In interviews, it's good to demonstrate which service type fits public, private, and hybrid networking needs.

2.3.4 What is Ingress in Kubernetes? How does it relate to Service? Can it support canary releases?

Ingress is a Kubernetes resource that manages HTTP(S) routing rules and acts as a Layer 7 gateway. It routes traffic to backend Services based on domain or path rules. Paired with Ingress Controllers like Nginx or Traefik, it supports TLS termination, rewriting, and canary deployments. Compared to Service, Ingress is more suited for centralized routing, API gateway patterns, and secure entry points. Canary releases can be implemented via annotations or custom rules (e.g., cookie/header-based routing). Candidates should be familiar with how Ingress works with Deployment and Service to support advanced routing.

2.3.5 What’s the difference between ConfigMap and Secret? How do you achieve config hot reload?

ConfigMap is used for injecting plain configuration data (e.g., YAML, environment variables), while Secret handles sensitive data (e.g., passwords, tokens), stored in base64 encoding. Both can be mounted as Volumes or injected via environment variables. To support hot reload, use volume mounts with file watchers (e.g., inotify) or sidecar containers that detect changes and notify the main container. Security practices such as

RBAC should be used to restrict Secret access. Understanding config lifecycle and hot-reload mechanisms is important in managing environments effectively.

2.3.6 How does Kubernetes perform health checks? What are probes used for?

Kubernetes supports three probe types: livenessProbe (to detect unresponsive containers and trigger restarts), readinessProbe (to remove unready Pods from load balancers), and startupProbe (for containers that take longer to initialize). Probes can use HTTP, TCP, or command checks. Proper probe configuration prevents service outages and cascading failures. Candidates should mention the importance of probe intervals, failure thresholds, and avoiding false positives.

2.3.7 How does the Kubernetes Scheduler decide which node a Pod should run on?

The kube-scheduler evaluates resource requirements (CPU, memory), affinity rules, taints and tolerations, and node selectors to determine placement. The process includes filtering (eliminating ineligible nodes) and scoring (ranking eligible nodes). The highest-scoring node is selected. Advanced scheduling strategies include custom schedulers, GPU-aware scheduling, and topology-aware placement. A strong answer discusses how to optimize pod placement for workload isolation and performance.

2.3.8 What is a DaemonSet, and how does it differ from Deployment?

A DaemonSet ensures a copy of a Pod runs on every node (or a subset). It’s typically used for node-level services like log collection (e.g., Fluentd), monitoring (e.g., NodeExporter), or CNI plugins. Unlike Deployment, DaemonSet doesn’t support horizontal scaling or rolling replicas but is node-aware. Candidates should explain when to use DaemonSet vs. Deployment based on application roles.

2.3.9 How does Kubernetes achieve storage persistence? What’s the relationship between PVC and PV?

Kubernetes separates storage provisioning (PV) from storage claims (PVC). PVs are provided by cluster admins or dynamically provisioned via StorageClasses. PVCs are storage requests by users. K8s binds PVCs to appropriate PVs, decoupling workload from storage details. StatefulSet Pods often bind to dedicated PVCs for stable storage. Use cases include databases, file uploads, and cache persistence. Interview answers should cover dynamic provisioning and volume reuse strategies.

2.3.10 How do you implement multi-environment isolation and multi-tenant access control in K8s?

Namespaces provide logical resource isolation for different environments (dev, test, prod). RBAC enables role-based access control, defining fine-grained permissions through

Role/ClusterRole and RoleBinding/ClusterRoleBinding. This ensures that developers, testers, and admins have scoped access. A good system design includes separate namespaces per environment, limited role access, and auditing policies to manage compliance and security.

2.4 Nginx

Keyword

Nginx is a high-performance HTTP server and reverse proxy server that utilizes an event-driven, asynchronous, non-blocking architecture (e.g., epoll/kqueue). It is widely used for load balancing, static/dynamic separation, caching acceleration, and security control. Nginx supports various load balancing strategies (round-robin, weight, IP hash), reverse proxy (proxy_pass), static resource delivery, TLS/HTTPS protocols, URL rewrite, Gzip compression, and rate limiting, making it a core component in microservices as an API Gateway or edge gateway.

Reverse Proxy is one of Nginx's core features, acting as an intermediary between clients and backend servers. It forwards client requests to internal services and returns the response back. It hides server structure and enables routing, load balancing, and security control. Unlike a forward proxy, it is transparent to clients and is primarily server-side. For example, in web services (Java/PHP/Python), Nginx serves as the entry point that routes traffic based on paths or load strategy to backend applications.

Load Balancing allows Nginx to distribute client requests across multiple backend servers to increase throughput and fault tolerance. Strategies include round-robin, weight-based, IP hash, and least connections. Configuring multiple servers via the upstream module enables high-availability clusters. This is commonly used in e-commerce systems for services like product and search.

Static and Dynamic Separation is a performance optimization pattern in which Nginx serves static files (HTML, CSS, JS, images) directly and proxies dynamic requests to backend apps. Using location directives to route URIs reduces backend load and improves response time. For instance, a homepage can be served entirely through cached static resources.

Caching Mechanisms such as proxy_cache and fastcgi_cache allow Nginx to store backend responses locally to reduce load and increase throughput. Caching can be tuned per URI, headers, parameters, or expiration time. It's widely used in high-traffic pages like news articles or product details.

Rewrite / Redirect enables URL modification and redirection. rewrite internally rewrites URIs; return 301/302 performs client-side redirection. Combined with regex, it's useful for SEO, path mapping, and legacy URL compatibility. For example, converting /product.php?id=1 to /product/1 helps search indexing.

WebSocket Reverse Proxy is supported via proxy_http_version 1.1 and Upgrade/Connection headers. This enables Nginx to handle WebSocket connections for real-time apps such as chat systems, live dashboards, or support tools.

Rate Limiting and Concurrency Control via limit_req and limit_conn directives helps prevent abuse and traffic spikes. Limits can be per-IP, per-connection, or per-request. For example, limiting login attempts or SMS verification APIs to avoid brute-force attacks.

SSL / TLS and HTTPS Setup is configured by setting up .crt certificates and .key private keys. Use listen 443 ssl, enable HTTP/2, force HTTPS redirection, and configure TLS settings for secure transmission. HTTPS is standard in production for public services.

Modular Configuration using include, server, and location blocks allows structured, maintainable configuration. server defines a virtual host, location manages request routing, and include supports config reuse. This modularity supports large-scale deployments and CI/CD integration.

Static File Serving is a strength of Nginx. Using root or alias, it serves files directly with high efficiency. expires and cache-control headers help browsers cache resources, improving load time and reducing bandwidth.

Logging and Monitoring via access_log and error_log provides visibility into request traffic and runtime issues. Logs can be formatted, filtered, and analyzed with ELK or Prometheus+Grafana for operational insights, anomaly detection, and auditing.

OpenResty / Lua Extension allows extending Nginx with Lua scripts for dynamic gateway logic, A/B testing, access control, or rate limiting. OpenResty, built on Nginx, supports flexible APIs and is used in API gateways and CDN edge computing platforms.

Relationship with API Gateway is a frequent interview topic. Nginx itself can act as a basic API Gateway handling routing, load balancing, and rate limiting, but lacks visual management and service governance. Modern gateways like Kong or APISIX extend Nginx/Envoy with plugins, service discovery, and orchestration. Use Nginx directly for lightweight needs, and a full gateway for complex microservice governance.

2.4.1 What are the differences between Nginx and Apache?

Nginx uses an asynchronous, event-driven model with low memory usage and is optimized for handling high concurrency. Apache uses a multi-threaded or multi-process model, offering rich features but with higher resource costs under load. Nginx is better for reverse proxy and high-load scenarios, while Apache fits legacy applications and full-featured server environments.

2.4.2 How does Nginx implement load balancing? What algorithms are supported?

Nginx uses the upstream block to group backend servers. It supports round-robin (default), weight, IP hash, and least-connections strategies. Health checks can be configured to exclude failed nodes automatically.

2.4.3 How does Nginx handle high concurrency?

Nginx leverages epoll (on Linux) or kqueue (on BSD) with an asynchronous, non-blocking I/O model. A master-worker architecture lets each worker process thousands of connections efficiently without thread switching.

2.4.4 How do you configure static-dynamic separation in Nginx?

Use location blocks to serve static files (like .jpg, .css, .js) directly, while other paths are proxied using proxy_pass. For example, /static/ is served locally and /api/ is proxied to backend apps.

2.4.5 How do you configure HTTPS in Nginx?

You must configure listen 443 ssl and provide the certificate and key files. Redirect all HTTP to HTTPS using return 301. Enable HTTP/2 and TLS tuning for security and performance.

2.4.6 How does Nginx implement access control?

You can use allow/deny rules, Basic Auth, User-Agent filtering, or Lua/JWT-based access validation. It's used to secure sensitive APIs or internal services.

2.4.7 How does Nginx perform rate limiting?

Use limit_req_zone and limit_req for request rate limiting (e.g., 10 requests/sec/IP), and limit_conn_zone with limit_conn to restrict concurrent connections.

2.4.8 What logging does Nginx support and how is it configured?

access_log records request metadata; error_log captures runtime errors. Format, level, and destination can be customized. Logs integrate with ELK/Grafana for real-time monitoring.

2.4.9 How does Nginx support WebSocket proxying?

Set proxy_http_version 1.1 and appropriate headers like Upgrade and Connection to forward WebSocket traffic. Used in real-time systems like chats and dashboards.

2.4.10 How do you optimize static asset caching in Nginx?

Use expires and cache-control headers in location blocks. Add file name versioning (e.g., hashed filenames) to control cache invalidation. This improves load speed and lowers server bandwidth.

2.5 API Gateway

Keyword

An **API Gateway** is the unified entry point in a microservices architecture. It handles request routing, authentication, rate limiting, circuit breaking, protocol translation, logging, and security. Popular gateways include Kong, Spring Cloud Gateway, Istio Gateway, and Nginx. Gateways forward client requests to internal services based on routing rules, while supporting CORS, JWT authentication, dynamic registration, and API versioning. API Gateways enhance maintainability and security and are central to service governance.

Routing is the fundamental function of an API Gateway, directing incoming client requests to backend services based on URI paths, methods, headers, and more. It supports static routes, dynamic paths with variables or wildcards, and regex-based routing. For instance, /api/v1/order can be routed to the order service, while /api/v1/user goes to the user center. This hides backend changes from the frontend and reduces coupling.

Protocol Translation allows the gateway to bridge different protocols. Clients may use HTTP/REST while backend services use gRPC, GraphQL, or WebSocket. The gateway handles encoding/decoding between JSON and Protobuf, connection conversions, and header-metadata mapping. This is common when HTTP clients interface with high-performance gRPC services or legacy systems require REST compatibility.

Authentication & Authorization are core security features of the API Gateway. It validates user identity and checks permissions before forwarding requests. Common mechanisms include JWT, OAuth2, API Keys, and sessions. In e-commerce or financial systems, JWTs encode user identity and permissions and are validated by the gateway before request processing.

Rate Limiting helps maintain system stability by restricting the frequency of requests per IP, user, or endpoint. Techniques include fixed window, sliding window, token bucket, and leaky bucket. Redis or in-memory counters implement these controls efficiently. Use cases include protecting SMS or login APIs from abuse.

Circuit Breaking & Fallback mechanisms detect downstream service failures and prevent cascading errors. If a backend returns errors or exceeds latency thresholds, the gateway triggers a circuit break and returns a default response. This keeps the system resilient. For instance, if the recommendation engine is down, return default product suggestions.

Service Discovery enables the API Gateway to detect backend services dynamically, often integrated with platforms like Consul, Eureka, or Kubernetes. Services auto-register at startup, and the gateway updates routing accordingly. In Kubernetes, this works with Ingress and CoreDNS to forward requests without manual configuration.

Plugin Architecture allows users to extend gateway functions with plugins—for logging, header injection, CORS, authentication, IP blacklisting, traffic mirroring, etc. Plugins support hot reloading and per-route configurations, enabling rapid deployment of observability, A/B testing, or security rules.

API Aggregation / Composition combines responses from multiple services into a single endpoint. This reduces frontend calls and is useful for bandwidth-constrained or unstable networks like mobile or IoT. For example, a homepage request can aggregate ads, user info, and product feeds into one response.

Multi-Tenancy Support ensures isolation of different teams or organizations. It includes route prefixes, custom auth, rate limits, and plugin policies per tenant. This is essential in SaaS platforms where multiple business units share infrastructure securely.

Observability in API Gateways includes access logs, tracing, metrics, and visualizations. Integrations with Prometheus, Grafana, ELK, or Zipkin provide real-time monitoring for QPS, latency, error rates, and throttling. These tools are crucial for SLA assurance, debugging, and capacity planning.

2.5.1 What is the role of an API Gateway? How does it differ from a reverse proxy?

An API Gateway is the core component in microservices that acts as a centralized access point. It handles routing, auth, rate limiting, logging, and protocol bridging. While reverse proxies focus on forwarding traffic, API Gateways offer richer features focused on API lifecycle and service governance.

2.5.2 How do you implement user authentication and access control in an API Gateway?

Gateways typically use JWT, OAuth2, or API Keys to validate user identity. After intercepting a request, the gateway checks token validity and expiration before forwarding. Role-based access can restrict access by path, method, or user group—essential for internal API protection.

2.5.3 How does an API Gateway perform dynamic rate limiting?

Rate limiting is implemented via token bucket or leaky bucket algorithms. Limits can be applied per IP, user ID, or endpoint. Redis is often used for counters. Plugins enable rule updates without downtime. Use cases include preventing abuse of SMS or flash-sale APIs.

2.5.4 How does the gateway handle circuit breaking and fallback?

If a downstream service fails or becomes slow, the gateway triggers a circuit breaker to short-circuit calls and prevent overload. Fallback returns cached or default responses to maintain UX. This prevents systemic failures due to a single service outage.

2.5.5 How is protocol translation handled in a gateway? What protocols are supported?

Modern gateways support HTTP, HTTPS, WebSocket, gRPC, and GraphQL. Protocol translation means accepting requests in one protocol (e.g., HTTP) and forwarding them as another (e.g., gRPC). Gateways also convert JSON ↔ Protobuf and manage headers. Useful in heterogeneous tech stacks.

2.5.6 How do gateways handle CORS (Cross-Origin Resource Sharing)?

Gateways respond with Access-Control-Allow-Origin, Methods, and Headers to enable cross-domain browser access. Preflight (OPTIONS) support is required. CORS can be configured via plugins or inline middleware and is critical in front-end–backend interactions.

2.5.7 What is API aggregation and why is it useful?

API aggregation merges multiple backend service calls into one response. This minimizes roundtrips and speeds up rendering on clients, especially on mobile or IoT. For instance, a dashboard page can aggregate product, banner, and user APIs into one call.

2.5.8 How does an API Gateway support gray release or A/B testing?

Gateways use routing strategies based on user ID, cookies, headers, or IP to segment traffic. Requests are directed to specific service versions. Combined with versioning and traffic mirroring, this allows controlled rollout and experimentation.

2.5.9 How does an API Gateway integrate with service registries?

Gateways like Kong or Spring Cloud Gateway can integrate with registries (e.g., Eureka, Consul, Nacos, Kubernetes) to auto-discover services. This allows real-time routing without hardcoding backend addresses, improving elasticity and operations.

2.5.10 How is tenant isolation achieved in multi-tenant API Gateways?

Isolation is achieved using path prefixes (/tenant-a/**), header-based routing, and token-bound tenant IDs. Each tenant can have its own rate limit, plugin config, and auth policy. This enables logical isolation in shared infrastructure for SaaS models.

2.6 Docker

Keyword

Docker is an open-source container platform that packages applications and their dependencies into lightweight, portable images, which can then run in isolated environments called containers. It leverages Linux kernel technologies such as **Namespaces** and **Cgroups** to achieve process-level isolation, significantly improving resource utilization and deployment flexibility. Docker images are layer-based, enabling rapid builds, caching, and cross-platform migration. Containers start in milliseconds and are ideal for dev/test isolation, CI/CD, microservice deployment, and stateless service execution.

Image: A Docker image is a read-only executable package that contains everything needed to run an application—OS libraries, runtime, environment variables, and source code. It is constructed via a Dockerfile and supports layered caching. Common commands include docker build, docker pull, and docker push. Multiple containers can be created from the same image.

Container: A container is a runtime instance of an image. It has its own file system, process space, and network stack, while sharing the host kernel. Containers are lightweight, isolated, and ephemeral. Commands include docker run, docker ps, and docker exec. Containers are suitable for stateless services or modular components in distributed systems.

Dockerfile: A script that defines how a Docker image is built, using directives like FROM, RUN, COPY, CMD, etc. A well-written Dockerfile improves build speed, reduces image size, and enhances maintainability. It is widely used in custom image creation and automated DevOps pipelines.

Volume: A mechanism for persistent storage in containers. Volumes outlive containers and enable data sharing between containers or with the host. Managed via -v flags or docker volume commands, types include anonymous volumes, named volumes, and bind mounts. Ideal for storing database data, logs, or shared content.

Docker Compose: A tool for defining and running multi-container Docker applications using a docker-compose.yml file. Services, networks, and volumes are declared declaratively. With a single docker-compose up, a full microservice stack can be launched—ideal for development, integration testing, or simplified local environments.

Registry: A service for storing and distributing Docker images. Docker Hub is the default public registry; private registries like Harbor are used in enterprise environments. Key commands include docker login, docker pull, and docker push.

Namespace / Cgroups: Core Linux features underpinning container isolation. Namespaces (e.g., PID, NET, IPC) isolate environment views, while Cgroups restrict CPU, memory, and other resource usage. Understanding these is fundamental to grasping container internals and system-level isolation.

Bridge / Host / Overlay Network:

- bridge (default): each container gets a virtual interface and routes through a bridge;
- host: container shares the host's network namespace—used for high-performance use cases;
- overlay: spans multiple hosts via VXLAN, used in Swarm or Kubernetes clusters. Network mode determines container connectivity and service reachability.

OCI & Runtime: Docker follows the **Open Container Initiative (OCI)** standard for image format and container runtime behavior. The default runtime is containerd + runc, but others like gVisor (runsc) are supported for enhanced security. Understanding OCI is key to working with Podman, CRI-O, and Kubernetes CRI runtimes.

2.6.1 What is the fundamental difference between Docker and virtual machines?

Docker containers run isolated processes directly on the host kernel, resulting in faster startup and lower overhead. In contrast, virtual machines run a full OS stack via a hypervisor, providing stronger isolation at the cost of higher resource usage. Containers provide lightweight process isolation, whereas VMs offer full system virtualization.

2.6.2 What are Docker images and containers? How are they related?

An image is a read-only template containing the application and runtime environment. A container is a mutable, running instance of an image, with its own filesystem and network stack. Multiple containers can share a single image, but their execution state and data are isolated.

2.6.3 What are common Dockerfile instructions? How can image build be optimized?

Typical commands include FROM, RUN, COPY, ENV, CMD, and EXPOSE.
Optimization tips:

- Combine multiple RUN commands to reduce layers;
- Use minimal base images like alpine;
- Remove unnecessary build dependencies;
- Use .dockerignore to exclude irrelevant files.

2.6.4 How do containers communicate with each other or the host?

Containers communicate via Docker networks. In `bridge` mode, containers can resolve each other by name. Use `docker network create` to set up a custom bridge network for internal service discovery. Port mapping (e.g., `-p 8080:80`) enables host-to-container access.

2.6.5 How is data persisted in Docker? What's the difference between Volume and Bind Mount?

Use **volumes** or **bind mounts**:

- Volumes are managed by Docker and ideal for portability, backups, and databases.
- Bind mounts map a host directory directly—more flexible but riskier.
  Use volumes for reliable persistence and isolation.

2.6.6 What is multi-stage build and what are its benefits?

Multi-stage builds use multiple `FROM` statements to separate build and runtime environments. Compile or package in an initial stage, and copy only the final artifact into the production image. Benefits: smaller images, improved security, and clearer build logic.

2.6.7 How do you debug a running container?

Use docker exec -it <container> /bin/sh to enter the container. Use docker logs, docker inspect, and docker stats to check output, metadata, and resource usage. Enable --privileged if low-level debugging is needed (with caution).

2.6.8 How do you apply resource limits in Docker?

You can limit memory and CPU via flags like --memory, --cpus, and --cpuset-cpus. For example: **docker run --memory=512m --cpus=1.5 myimage**

This limits the container to 512MB memory and 1.5 CPU cores. Limits are enforced using Cgroups.

2.6.9 What networking modes does Docker support and when to use them?

- bridge (default): isolated container-to-host communication;
- host: shares host network stack (faster, but less isolated);
- none: no network assigned;
- overlay: for multi-host networking (e.g., Swarm/K8s).
  Choose based on performance, security, and scalability needs.

2.6.10 What are common Docker security risks and how can they be mitigated?

Risks include privilege escalation, image backdoors, secret leakage, and volume compromise.
Mitigation strategies:

- Use verified image sources;
- Run containers as non-root users with --user;
- Drop unnecessary capabilities with --cap-drop;
- Avoid --privileged;
- Enable AppArmor or Seccomp profiles;
- Use scanners like **Trivy** for image vulnerability audits.

3. Cloud Computing Platforms and Modern Infrastructure Services

Cloud computing is typically divided into three core service models: **IaaS (Infrastructure as a Service)**: Provides underlying infrastructure such as virtual machines, storage, and networking. Examples include AWS EC2 and Alibaba ECS. **PaaS (Platform as a Service)**: Offers a platform and environment for deploying applications without managing underlying hardware. Examples include Google App Engine and Heroku. **SaaS (Software as a Service)**: Provides fully functional software applications to users, which can be accessed over the internet. Examples include Salesforce and Google

Workspace. Understanding the differences between these models helps in making appropriate architecture decisions in real-world applications.

Leading cloud providers like AWS, Azure, Google Cloud, Alibaba Cloud, and Tencent Cloud offer complete cloud computing services, including computing (ECS, Fargate), storage (S3, OSS), databases (RDS, DynamoDB), containers (EKS, AKS, ACK), Serverless, AI services, and more. These platforms support global deployment, elastic resource scheduling, and pay-as-you-go pricing, making them the primary choice for modern architecture deployments.

Object storage utilizes a key-value structure, suitable for managing large-scale unstructured data. Services like **Amazon S3**, **Alibaba OSS**, and **Google Cloud Storage** provide features like cold and hot data tiering, version control, pre-signed URLs, and bucket policies, ideal for backup, archiving, and data lake storage.

A **CDN** caches static resources at global edge nodes, improving access speed and bandwidth optimization. Major providers like **Cloudflare**, **Akamai**, and cloud-native CDN services from Alibaba and Tencent allow integration with security features such as anti-leeching, IP blacklisting, WAF (Web Application Firewall), and HTTPS forwarding. CDNs are used for accelerating frontend resources, API rate-limiting, and streaming media distribution.

Serverless computing emphasizes "no maintenance, on-demand execution," with **FaaS** (Function as a Service) like **AWS Lambda** or **Alibaba Cloud Function Compute**, where functions scale automatically and are billed per invocation. **BaaS** (Backend as a Service) such as **Firebase** and **Supabase** provide integrated services like authentication, databases, and storage. Serverless architecture is ideal for event-driven services and handling burst traffic.

Many enterprises use multi-cloud or hybrid cloud strategies to enhance stability and negotiating power. Multi-cloud deployment avoids vendor lock-in and provides geographical fault tolerance. Hybrid cloud combines private and public clouds, fulfilling compliance and sensitivity requirements. Tools like **Terraform** (Infrastructure as Code) and **KubeFed** (Kubernetes Federation) support these strategies.

Leading cloud platforms offer native monitoring services (e.g., **AWS CloudWatch**, **Alibaba Cloud Monitoring**, **Tencent Cloud Monitoring**) to track resource usage, service health, and API call analysis. Tools like **Prometheus** and **Grafana** allow customized metric collection and visualization, and **AlertManager** or **DingTalk** robots can be used for fast failure notifications.

Cloud-native data warehouses like **Snowflake** and **Google BigQuery** separate compute and storage to provide elastic scalability, SQL querying, high concurrency, and real-time analysis. **Cloud-native data lakes** such as **AWS Lake Formation** and **Databricks Delta Lake** support ACID transactions, schema evolution, and streaming writes, facilitating modern data warehouse construction.

Cloud environments emphasize **Zero Trust Architecture**, where no access is trusted by default. Security is ensured using IAM (Identity and Access Management) models, multi-factor authentication, VPC network isolation, VPNs, bastion hosts, TLS encryption, and firewall policies. **Cloudflare Zero Trust** and **AWS IAM Policy** are examples of such systems.

4. Designing a High-Availability Logging System

(1). Log Collection Layer. The first layer of the **logging system** is the collection layer, deployed close to the business services at edge nodes. Lightweight agents like **Filebeat** or **Fluent Bit** collect logs from local log files or container stdout. The log collector should support time-based rotation, log splitting by key fields, and include sampling and filtering mechanisms to control log volume at the source.

(2). Log Transmission Layer. Collected logs are sent asynchronously and in batches to a **Kafka** cluster. Kafka's partitioning mechanism supports concurrent producers while using replication to ensure persistence. With the **ISR (in-sync replicas)** mechanism, data is protected against node failures. Local buffers or **Redis** can be used as intermediary caches to prevent bottlenecks during writing.

(3). Log Processing Layer. Logs stored in Kafka are consumed by multiple asynchronous consumers forming an independent **Log Processor** service cluster. Processing logic includes tasks such as JSON parsing, field normalization, sensitive data masking, labeling, and filtering out invalid logs. The processed data is then written into back-end systems like **Elasticsearch**, **ClickHouse**, or **InfluxDB** for multi-dimensional search and analysis.

(4). Log Storage and Query Layer. Log storage should employ a cold and hot layer strategy. Recent logs (hot data) can be stored in high-performance query databases like ClickHouse, while historical logs (cold data) are archived in HDFS, S3, or object storage. Reverse proxy through Nginx can optimize authentication and caching. Tools like Kibana and Grafana can be integrated for user-facing visualizations.

(5). High Availability and Disaster Recovery Design. The entire logging pipeline should be designed with disaster recovery and redundancy capabilities. **Kafka** supports partition replication, and the consumer layer should deploy replicas with distributed coordination (e.g., **ZooKeeper** or **Etcd**). **Log Processor** should support automatic restarts and state recovery. Anomaly monitoring systems should detect writing failures, backlog consumption, and failed queries in **Elasticsearch**, alerting promptly and allowing log backfilling and delay metrics display.

5. Designing Scalable Data Analysis Architecture

(1). Data Ingestion Layer. The analysis system should support the ingestion of heterogeneous data sources, including API pulls, Change Data Capture (CDC), Kafka topic subscriptions, file uploads, etc. A unified access platform or middleware (e.g., Apache NiFi) can be used for data protocol conversion, structure standardization, and permission control.

(2). Data Transmission Layer. To handle large-scale data flow, Kafka or Pulsar are recommended as distributed messaging systems for buffering and decoupling. Data is partitioned by business line or functionality, and consumer groups are used to scale horizontally and increase throughput.

(3). Real-time Computation and Preprocessing Layer. Real-time data streams should be processed by Flink or Spark Streaming for log cleaning, aggregation, metric generation, tagging, and joining with dimension tables in ETL operations. This layer should support state fault-tolerance mechanisms like checkpoints or state snapshots to ensure computational consistency.

(4). Offline Batch Processing Layer. Historical data analysis can be triggered on a schedule by orchestration systems like Airflow or DolphinScheduler, running batch SQL jobs via engines like Hive, Presto, or Trino. Data lakes (e.g., Hudi, Iceberg, Delta Lake) should support incremental merging and schema evolution.

(5). Data Storage Layer. Data should be stored using a hot and cold layer strategy. Hot data can be stored in high-concurrency databases like **ClickHouse** and **Druid** for near-real-time queries. Cold data is archived in low-cost object storage like **HDFS** or **S3**. The system should support data partitioning, compression, and lifecycle management.

(6). Query and Service Interface Layer. A unified query API gateway should be exposed, supporting parameter validation, rate limiting, and access control. Integration with Superset, Metabase, or Grafana for user self-service querying and visualization is essential. Complex queries should have orchestration and result caching mechanisms to improve performance.

(7). Orchestration and Monitoring Layer. Task execution should be managed by a unified orchestration platform, recording execution statuses and dependencies, supporting reruns, failure alerts, and version control. The platform should have metrics collection, delay alerts, data validation, and lineage tracking capabilities to ensure the stability and consistency of the entire analysis system.

Popular Development Tools and Engineering Practices

GDB is a command-line debugger widely used in C/C++ for debugging program crashes, logic errors, and performance bottlenecks. Common commands like break, step/next, print/info, and x/ help analyze code flow and memory structure. GDB supports core dump analysis and remote debugging, which are essential for system-level and low-level development.

Valgrind is commonly used on Linux for detecting memory leaks, undefined behaviors, and illegal reads/writes. Its memcheck module prints detailed stack trace information.

AddressSanitizer (ASan) and LeakSanitizer (LSan) provide high-performance detection, easily integrated into CI processes for real-time checks. They are important tools for maintaining high-quality C/C++ code.

Git is the most popular distributed version control system, supporting branching, staging, rebasing, interactive commits, and more. GitHub and GitLab offer hosting, collaboration, CI/CD integration, and issue tracking, forming the complete development workflow. Mastery of Git and resolving merge conflicts are essential for team collaboration.

SVN is a centralized version control system still used in traditional software development processes and environments with more restrictive networks. Understanding SVN's branch/merge strategy and conflict resolution is valuable for working in legacy systems.

CMake is a modern, cross-platform build system configuration tool for C++ projects, managing complex builds across different platforms. Make is a lower-level build tool suitable for smaller projects or embedded development environments.

Clang and GCC are the most commonly used compilers for C/C++. Mastering compiler flags like -O2/-O3/-g/-fsanitize/-Wall helps adjust the build process for debugging, performance optimization, and testing.

Jenkins is an old but still popular CI platform, supporting pipeline definitions for building, testing, and deploying software. GitHub Actions and GitLab CI provide modern, easy-to-use CI/CD configurations with integrations into containers.

Docker provides lightweight environment isolation, supporting the entire lifecycle from code packaging to deployment. Developers use Dockerfile to build images and Compose for service orchestration, allowing easy local simulation and cross-platform deployments.

Ccache improves incremental compilation efficiency by caching precompiled results, ideal for large projects. Distcc distributes compilation tasks to multiple hosts, significantly speeding up full builds.

These are popular logging libraries for C++, Log4cxx / spdlog / glog. They offer multi-level log outputs (INFO/WARN/ERROR/DEBUG), asynchronous writes, and rolling files, with different use cases and performance optimizations (like spdlog being lightweight and high-performance).

Wireshark is a powerful network protocol analyzer with a graphical interface, suitable for detailed packet inspection, session reconstruction, and protocol decoding. Tcpdump is a lightweight command-line tool ideal for remote network diagnostics.

Cppcheck is designed to detect basic errors in C/C++ code, such as null pointer dereferencing.

Clang-Tidy supports custom rule sets and modern C++ code style checks.

SonarQube helps with code quality monitoring, supporting multiple languages, coverage metrics, and technical debt tracking in team environments.

Common SQL Commands(For MySQL/PostgreSQL)

1. Database Operations:

-- Create Database

```
CREATE DATABASE db_name;
```

-- Drop Database

```
DROP DATABASE db_name;
```

-- Use Database

```
USE db_name;
```

2. Table Operations:

-- Create Table:

```
CREATE TABLE users (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(50),
  age INT,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

-- View Table Structure:

```
DESC users; -- MySQL
\d users;   -- PostgreSQL
```

-- Add Column:

```
ALTER TABLE users ADD COLUMN email VARCHAR(100);
```

-- Modify Column Type:

```
ALTER TABLE users ALTER COLUMN age TYPE BIGINT; -- PostgreSQL
```

-- Drop Column:

```
ALTER TABLE users DROP COLUMN email;
```

-- Drop Table:

```
DROP TABLE users;
```

3. Inserting Data:

```
INSERT INTO users (name, age) VALUES ('Alice', 25);
INSERT INTO users VALUES (NULL, 'Bob', 30, DEFAULT);
```

4. Querying Data:

-- Basic Query:

```
SELECT * FROM users;
```

-- Conditional Query:

```
SELECT * FROM users WHERE age > 20 AND name LIKE 'A%';
```

-- Sorting + Limiting:

SELECT * FROM users ORDER BY age DESC LIMIT 10;

-- Aggregation:

SELECT COUNT(*) FROM users;

SELECT AVG(age), MAX(age), MIN(age) FROM users;

-- Grouping:

SELECT age, COUNT(*) FROM users GROUP BY age;

-- Grouping + Condition:

SELECT age, COUNT(*) FROM users GROUP BY age HAVING COUNT(*) > 1;

5. Updating & Deleting Data:

-- Update Data:

UPDATE users SET age = age + 1 WHERE name = 'Alice';

-- Delete Data:

DELETE FROM users WHERE age < 18;

-- Clear Table Data (Without Dropping Structure):

TRUNCATE TABLE users;

6. Joins:

-- Inner Join:

SELECT u.name, o.amount

FROM users u

JOIN orders o ON u.id = o.user_id;

-- Left Join (Preserves Left Table Rows):

SELECT u.name, o.amount

FROM users u

LEFT JOIN orders o ON u.id = o.user_id;

-- Subquery:

```
SELECT * FROM users WHERE id IN (
  SELECT user_id FROM orders WHERE amount > 100
);
```

7. Indexes, Views & Transactions::

-- Create Index:

```
CREATE INDEX idx_name ON users(name);
```

-- Create View:

```
CREATE VIEW adult_users AS
SELECT * FROM users WHERE age >= 18;
```

-- Transaction Control:

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
```

-- Rollback:

```
ROLLBACK;
```

Common Linux Commands

1. File/Directory Operations:

```
ls -al          # List all files (including hidden)
cd /path/to/   # Change directory
mkdir logs     # Create directory
rm -rf logs    # Remove directory/file
```

cp file1 file2 # Copy file

mv a b   # Rename/Move file

touch a.txt   # Create empty file

2. View File Contents:

cat file.txt   # View full content

less file.txt   # View content with pagination

tail -n 50 file.txt   # View last 50 lines

tail -f file.txt   # Real-time log tracking

head -n 10 file.txt   # View first 10 lines

3. Search & Text Processing:

grep "error" app.log   # Search for 'error' in app.log

grep -r "TODO" ./src   # Recursively search for 'TODO' in src directory

find . -name "*.log"   # Find all .log files

awk '{print $1, $3}' data.txt   # Print columns 1 and 3 of data.txt

sed 's/old/new/g' file.txt   # Replace 'old' with 'new' in file.txt

cut -d ',' -f 1,2 users.csv   # Split CSV by comma and print columns 1 and 2

4. Permissions & User Info:

chmod +x script.sh   # Add executable permission

chown user:group file   # Change file owner

whoami   # Current logged-in user

id   # Current user UID/GID

5. System Information & Processes:

```
top / htop            # Real-time system monitoring

ps aux | grep nginx     # View specific process

kill -9 PID            # Force kill process by PID

df -h                 # View disk space

du -sh /path          # View directory size

free -m               # View memory usage

uptime                # System uptime

uname -a              # View kernel info
```

6. Networking Commands:

```
ping google.com              # Test connectivity

curl http://localhost:8080    # Send HTTP request

netstat -tulnp               # View listening ports (older systems)

ss -tulnp                    # Replacement for netstat

lsof -i:8080                 # View processes using port 8080

ifconfig / ip addr           # View network interface info

traceroute 8.8.8.8           # Trace route to IP
```

7. Development Tools:

```
gcc main.c -o app           # Compile C program

g++ main.cpp -o app          # Compile C++ program

make / make clean           # Build/Clean project
```

```
tar -czvf app.tar.gz app/     # Compress directory

scp file user@host:/path      # Remote copy file

ssh user@host                 # Remote login

crontab -e                    # Edit scheduled tasks
```

8. Logs & Service Management:

```
journalctl -u nginx.service --since "1 hour ago"   # View logs for nginx service in the last hour

systemctl status nginx         # View service status

systemctl restart nginx        # Restart service

systemctl enable nginx         # Enable service at boot
```

Common development and debugging tool commands

1. Common Git Commands

1. Repository Initialization & Cloning:

```
git init                               # Initialize a local Git repository

git clone https://github.com/user/repo.git   # Clone a remote repository
```

2. Status & Logs:

```
git status                             # Show current working status

git log                                # View commit history

git log file.cpp                       # View commit history for a specific file

git diff                               # Show unstaged changes
git diff --cached                    # Show staged changes (ready to commit)

git diff --stat                      # Show file-level change summary
```

3. Staging & Committing:

```
git add .                            # Stage all modified files

git add main.cpp                     # Stage a specific file

git commit -m "commit message"       # Commit changes with message

git commit --amend -m "new message"  # Modify the last commit message
```

4. Branch Management:

```
git branch                           # List local branches

git branch dev                       # Create a new branch

git checkout dev                     # Switch to an existing branch

git checkout -b new-feature          # Create and switch to a new branch

git merge dev                        # Merge a branch into the current branch

git branch -d old-feature            # Delete a local branch
```

5. Remote Operations:

```
git remote -v                        # List configured remote repositories

git remote add origin URL            # Add a remote repository

git push -u origin main              # Push local code to remote (with tracking)

git pull                             # Fetch and merge remote changes

git fetch                            # Fetch remote changes without merging
```

6. Undo & Reset:

git checkout -- file.cpp # Discard local changes to a file

git reset HEAD file.cpp # Unstage a file (move back to working dir)

git reset --soft HEAD^ # Roll back to the previous commit (keep staged changes)

git reset --mixed HEAD^ # Roll back to the previous commit (keep working dir changes)

git reset --hard HEAD^ # Roll back and discard all changes

git rm --cached file.cpp # Remove a file from version control (keep locally)

7. .gitignore Configuration:

*.log

*.o

/build/

.DS_Store

8. User Info & Git Identity:

git config --list # List all Git config settings

git config user.name # Show current username

git config user.email # Show current email

git config --global user.name "xxx" # Set global username

git config --global user.email "xxx" # Set global email

2. GDB Command Reference

1. Basic Debugging and Breakpoint Setup:

gdb ./a.out # Start GDB and load the program

gdb ./a.out core # Load a core dump file for crash analysis

```
break main                  # Set a breakpoint at the main function
break file.cpp:42           # Set a breakpoint at line 42 of file.cpp
break func if var == 3      # Conditional breakpoint (triggered if var == 3)
delete                      # Delete all breakpoints
run                         # Run the program
```

2. Step Execution and Program Control:

```
next / n                    # Step over (do not enter functions)
step / s                    # Step into functions
finish                      # Run until the current function returns
continue / c                # Continue execution until the next breakpoint
until 88                    # Continue until reaching line 88
```

3. Variable Inspection and Expression Evaluation:

```
print var                   # Print the value of a variable
print *ptr                  # Dereference and print the value pointed to
display var                 # Automatically display a variable on each step
set var = 123               # Modify the value of a variable
```

4. Stack Frame and Call Chain Analysis:

```
backtrace / bt              # Display the current call stack
backtrace full              # Display full local variables for each frame
frame 0                     # Switch to frame 0
info locals                 # Show local variables in the current frame
info args                   # Show function arguments of the current frame
```

5. Memory and Register Debugging:

```
x/4x &var                    # Examine memory at an address (4 hex words)

x/s str                      # Print the string at a given address

info registers               # View CPU register contents
```

6. Multi-threaded Debugging:

```
info threads                 # List all threads

thread 3                     # Switch to thread number 3

bt                           # Show call stack for the current thread
```

7. Core Dump Debugging (Linux):

```
ulimit -c unlimited          # Enable core dump generation

echo "/tmp/core.%e.%p" > /proc/sys/kernel/core_pattern  # Set core dump file pattern

./a.out                      # Run the program until it crashes (generating core)

gdb ./a.out core.xxxx        # Load the core dump file for analysis

bt                           # Show backtrace (call stack) at crash time

info registers               # Show register values at crash
```

Error Type	Detection Method	Troubleshooting Tips
Segmentation fault	bt + info locals	Check for null pointers, illegal memory access, or array out-of-bounds.
Bus error	Typically occurs due to misaligned access or invalid hardware address	Check address alignment and memory mapping.
double free / invalid free	Crash occurs during a free() call	Set a breakpoint on free to inspect the call stack.
use after free	bt + print the accessed memory address	Use **Valgrind** to check if memory was freed too early.

stack smashing	GDB reports __stack_chk_fail	Check for buffer overflows, especially on the stack.

3. Windbg Command Reference

1. Load Dump File:

```
windbg -z myapp.dmp                    # Open the .dmp file

.sympath srv*C:\Symbols*https://msdl.microsoft.com/download/symbols

.reload                                # Load symbols
```

2. Automatic Crash Analysis:

```
!analyze -v                            # Analyze the crash cause and call stack

!analyze -show                         # Display the last analysis result
```

3. View Call Stack and Symbols:

```
kv                                     # Display the call stack with arguments

kb                                     # Display a simple call stack

~* kb                                  # Show call stacks for all threads

lm                                     # List all loaded modules
```

4. Register and Variable Information:

```
r                                      # Show CPU register values

dv                                     # Display local variables of the current stack frame

dt module!StructName                   # View structure fields of a specific type
```

5. Memory and Heap Analysis:

```
!heap -s                        # Heap summary

!heap -stat                      # Show allocation statistics by size

!heap -p -a <addr>                    # Find which heap block an address belongs to

!address <addr>                      # Get information about a specific address
```

6. Set Breakpoints and Control Execution (Live Process Debugging):

```
bp kernel32!CreateFileW                # Set a breakpoint on CreateFileW

g                                 # Continue program execution
```

Error Code	Meaning	**Troubleshooting Suggestions**
0xc0000005	Access violation	Use !analyze -v and kv to examine the crash and call stack.
STACK_OVERFLOW	Stack overflow	Check for deep or infinite recursion and abnormal function arguments.
HEAP_CORRUPTION	Heap corruption	Use !heap -s and !heap -p -a to locate the corrupted heap block.
INVALID_POINTER	Using uninitialized or freed memory	Check for **use-after-free** or uninitialized pointer access.

4. Valgrind / AddressSanitizer / LeakSanitizer Command Reference

1. Basic Valgrind Usage:

```
valgrind ./a.out                          # Run valgrind to check for memory issues

valgrind --leak-check=full ./a.out        # Perform full memory leak analysis

valgrind --track-origins=yes ./a.out      # Show the origin of uninitialized values

valgrind --log-file=valgrind.log ./a.out  # Output the report to a log file

valgrind --tool=memcheck ./a.out          # Specify the tool (memcheck is the default)
```

2. Common Valgrind Output Types:

```
valgrind ./a.out                          # Run valgrind to check for memory issues
```

Output Type	Meaning
Definitely lost	Memory is definitely leaked — no references exist
indirectly lost	Memory is leaked indirectly via pointers from leaked blocks
possibly lost	Memory might be leaked — pointers lost in complex structures
still reachable	Memory is still accessible at program exit(not necessarily a leak)

3. Switching Valgrind Tools:

```
valgrind --tool=memcheck                # Default tool for memory read/write checks

valgrind --tool=callgrind ./a.out         # Analyze function call frequency (performance profiling)

valgrind --tool=massif ./a.out           # Monitor heap memory usage over time

ms_print massif.out.pid                  # Visualize Massif memory usage report
```

4. AddressSanitizer Compilation and Execution:

```
g++ -fsanitize=address -g main.cpp -o main   # Compile with AddressSanitizer enabled

./main                                   # Run directly, memory issues will be reported automatically
```

5. LeakSanitizer Compilation and Execution:

```
g++ -fsanitize=leak -g main.cpp -o main      # Enable LeakSanitizer (included in AddressSanitizer by default)

ASAN_OPTIONS=detect_leaks=1 ./main          # Explicitly enable leak detection
```

6. Common Error Types (Valgrind / ASan Keywords):

Error Type	Explanation	Troubleshooting Tip
invalid read / write	Accessing invalid memory (e.g., out-of-bounds, freed ptr)	Check for array bounds, pointer validity
use-after-free	Accessing memory after it has been freed	Check deallocation logic, use Valgrind or ASan
memory leak	Heap memory allocated but not freed	Ensure proper delete or free at all code paths

uninitialized value	Using a variable before it is initialized	Initialize variables; use --track-origins=yes to trace the source of the uninitialized value.
stack-use-after-return	Accessing a local variable after function returns	Avoid returning pointers/references to local variables

5. Docker Command Reference (Build / Run / Debug)

1. Image Building and Management:

```
docker build -t myapp .                  # Build an image from the current directory

docker images                          # List all local images

docker rmi myapp                         # Remove a specific image

docker tag myapp myrepo/myapp:v1          # Tag an image for pushing to a repository
```

2. Container Execution and Control:

```
docker run -it myapp                     # Run a container interactively

docker run -d -p 8080:80 myapp            # Run in detached mode and map port 8080 to 80

docker exec -it container_id bash         # Enter a running container via bash

docker stop container_id                  # Stop a running container

docker rm container_id                    # Remove a stopped container
```

3. Container and Image Management:

```
docker ps / docker ps -a                  # List running containers / all containers

docker logs container_id                  # View logs from a container

docker inspect container_id                # Show detailed container configuration
docker system prune                    # Clean up unused containers, networks, and images
```

4. Sample Dockerfile:

```
FROM ubuntu:20.04

RUN apt update && apt install -y g++

COPY . /app

WORKDIR /app

CMD ["./main"]
```

5. Volume Mounting and Debugging Tips:

```
docker run -v $(pwd):/workspace -w /workspace ubuntu bash  # Mount current directory into container

docker run --rm -it myimage bash                    # Launch a disposable debug container
```