

Log In

Join

Back To Course Home

Grokking Modern System Design Interview for Engineers & Managers

0% completed

System Design Interviews

Introduction

Abstractions

Non-functional System Characteristics

Back-of-the-envelope Calculations

Building Blocks

Domain Name System

Load Balancers

Databases

Key-value Store

Content Delivery Network (CDN)

Sequencer

Distributed Monitoring

Monitor Server-side Errors

Monitor Client-side Errors

Distributed Cache

Distributed Messaging Queue

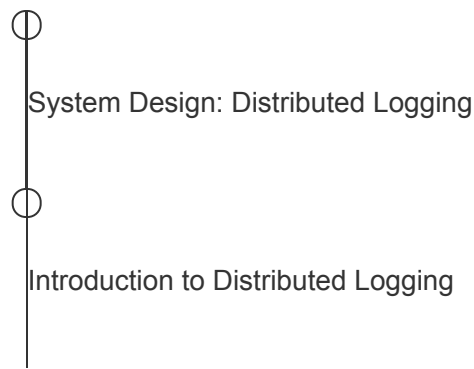
Pub-sub

Rate Limiter

Blob Store

Distributed Search

Distributed Logging



System Design: Distributed Logging
Introduction to Distributed Logging

Design of a Distributed Logging Service

Distributed Task Scheduler

Sharded Counters

Concluding the Building Blocks Discussion

Design YouTube

Design Quora

Design Google Maps

Design a Proximity Service / Yelp

Design Uber

Design Twitter

Design Newsfeed System

Design Instagram

Design a URL Shortening Service / TinyURL

Design a Web Crawler

Design WhatsApp

Design Typeahead Suggestion

Design a Collaborative Document Editing Service / Google Docs

Spectacular Failures

Concluding Remarks

Course Certificate

Mark Course as Completed

Design of a Distributed Logging Service

Learn how to design a distributed logging service.

We'll cover the following

- Requirements
 - Functional requirements
 - Non-functional requirements
- Building blocks we will use
- API design
- Initial design
- Logging at various levels
 - In a server
 - At datacenter level
- Conclusion

We'll design the distributed logging system now. Our logging system should log all activities or messages (we'll not incorporate sampling ability into our design).

Requirements#

Let's list the requirements for designing a distributed logging system:

Functional requirements#

The functional requirements of our system are as follows:

- **Writing logs:** The services of the distributed system must be able to write into the logging system.
- **Searchable logs:** It should be effortless for a system to find logs. Similarly, the application's flow from end-to-end should also be effortless.
- **Storing logging:** The logs should reside in distributed storage for easy access.
- **Centralized logging visualizer:** The system should provide a unified view of globally separated services.

Non-functional requirements#

The non-functional requirements of our system are as follows:

- **Low latency:** Logging is an I/O-intensive operation that is often much slower than CPU operations. We need to design the system so that logging is not on an application's critical path.
- **Scalability:** We want our logging system to be scalable. It should be able to handle the increasing amounts of logs over time and a growing number of concurrent users.
- **Availability:** The logging system should be highly available to log the data.

Building blocks we will use#

The design of a distributed logging system will utilize the following building blocks:

- **Pub-sub system:** We'll use a pub-sub- system to handle the huge size of logs.
- **Distributed search:** We'll use distributed search to query the logs efficiently.

Building blocks we will use

API design#

The API design for this problem is given below:

Write a message

The API call to perform writing should look like this:

```
write(unique_ID, message_to_be_logged)
```

Parameter	Description
unique_ID	It is a numeric ID containing application-id service-id, and a time stamp.
message_to_be_logged	It is the log message stored against a unique key.

Search log

The API call to search data should look like this:

```
searching(keyword)
```

This call returns a list of logs that contain the keyword.

Parameter	Description
keyword	It is used for finding the logs containing the keyword

Initial design#

In a distributed system, clients across the globe generate events by requesting services from different serving nodes. The nodes generate logs while handling each of the requests. These logs are accumulated on the respective nodes.

In addition to the building blocks, let’s list the major components of our system:

- **Log accumulator:** An agent that collects logs from each node and dumps them into storage. So, if we want to know about a particular event, we don’t need to visit

each node, and we can fetch them from our storage.

- **Storage:** The logs need to be stored somewhere after accumulation. We'll choose blob storage to save our logs.
- **Log indexer:** The growing number of log files affects the searching ability. The log indexer will use the distributed search to search efficiently.
- **Visualizer:** The visualizer is used to provide a unified view of all the logs.

The design for this method looks like this:

Initial design

There are millions of servers in a distributed system, and using a single log accumulator severely affects scalability. Let's learn how we'll scale our system.

Logging at various levels#

Let's explore how the logging system works at various levels.

In a server#

In this section, we'll learn how various services belonging to different apps will log in to a server.

Let's consider a situation where we have multiple different applications on a server, such as App 1, App 2, and so on. Each application has various microservices running as well. For example, an e-commerce application can have services like authenticating users,

fetching carts, and more running at the same time. Every service produces logs. We use an ID with `application-id`, `service-id`, and its time stamp to uniquely identify various services of multiple applications. Time stamps can help us to determine the causality of events.

Each service will push its data to the **log accumulator** service. It is responsible for these actions:

- Receiving the logs.
- Storing the logs locally.
- Pushing the logs to a pub-sub system.

We use the pub-sub system to cater to our scalability issue. Now, each server has its log accumulator (or multiple accumulators) push the data to pub-sub. The pub-sub system is capable of managing a huge amount of logs.

To fulfill another requirement of low latency, we don't want the logging to affect the performance of other processes, so we send the logs asynchronously via a low-priority thread. By doing this, our system does not interfere with the performance of others and ensures availability.



Multiple applications running on a server, and each application has various microservices

1 of 3

3 of 3 We should be mindful that data can be lost in the process of logging huge amounts of messages. There is a trade-off between user-perceived latency and the guarantee that log data persists. For lower latency, log services often keep data in RAM and persist them asynchronously. Additionally, we can minimize data loss by adding redundant log accumulators to handle growing concurrent users.

Point to Ponder

Question

How does logging change when we host our service on a multi-tenant cloud (like AWS) versus when an organization has exclusive control of the infrastructure (like Facebook), specifically in terms of logs?

Show Answer

Note: For applications like banking and financial apps, the logs must be very secure so hackers cannot steal the data. The common practice is to encrypt the data and log. In this way, no one can decrypt the encrypted information using the data from logs.

At datacenter level#

All servers in a data center push the logs to a pub-sub system. Since we use a horizontally-scalable pub-sub system, it is possible to manage huge amounts of logs. We may use multiple instances of the pub-sub per data center. It makes our system scalable, and we can avoid bottlenecks. Then, the pub-sub system pushes the data to the blob storage.

Log accumulator sending data to the pub-sub system

The data does not reside in pub-sub forever and gets deleted after a few days before being stored in archival storage. However, we can utilize the data while it is available in the pub-sub system. The following services will work on the pub-sub data:

- **Filterer:** It identifies the application and stores the logs in the blob storage reserved for that application since we do not want to mix logs of two different applications.
- **Error aggregator:** It is critical to identify an error as quickly as possible. We use a service that picks up the error messages from the pub-sub system and informs the respective client. It saves us the trouble of searching the logs.
- **Alert aggregator:** Alerts are also crucial. So, it is important to be aware of them early. This service identifies the alerts and notifies the appropriate stakeholders if a fatal error is encountered, or sends a message to a monitoring tool.

The updated design is given below:

Adding a filterer, error aggregator, and alert aggregator

Point to Ponder

Question

Do we store the logs for a lifetime?

Show Answer

In our design, we have identified another component called the **expiration checker**. It is responsible for these tasks:

- Verifying the logs that have to be deleted. Verifying the logs to store in cold storage.

Moreover, our components log indexer and visualizer work on the blob storage to provide a good searching experience to the end user. We can see the final design of the logging service below:

Logging service design

Point to Ponder

Question

We learned earlier that a simple user-level API call to a large service might involve

hundreds of internal microservices and thousands of nodes. How can we stitch together logs end-to-end for one request with causality intact?

Show Answer

Note: Windows Azure Storage System (WAS) uses an extensive logging infrastructure in its development. It stores the logs in local disks, and given a large number of logs, they do not push the logs to the distributed storage. Instead, they use a grep-like utility that works as a distributed search. This way, they have a unified view of globally distributed logs data.

There can be various ways to design a distributed logging service, but it solely depends on the requirements of our application.

Conclusion#

- We learned how logging is crucial in understanding the flow of events in a distributed system. It helps to reduce the mean time to repair (MTTR) by steering us toward the root causes of issues.
- Logging is an I/O-intensive operation that is time-consuming and slow. It is essential to handle it carefully and not affect the critical path of other services' execution.
- Logging is essential for monitoring because the data fetched from logs helps monitor the health of an application. (Alert and error aggregators serve this purpose.)

Back

Introduction to Distributed Logging

Next

System Design: The Distributed Task ...

Mark as Completed

Report an Issue