• Standard BFS does not take the priority of a URL into consideration. The web is large and not every page has the same level of quality and importance. Therefore, we may want to prioritize URLs according to their page ranks, web traffic, update frequency, etc.

URL frontier

URL frontier helps to address these problems. A URL frontier is a data structure that stores URLs to be downloaded. The URL frontier is an important component to ensure politeness, URL prioritization, and freshness. A few noteworthy papers on URL frontier are mentioned in the reference materials [5] [9]. The findings from these papers are as follows:

Politeness

Generally, a web crawler should avoid sending too many requests to the same hosting server within a short period. Sending too many requests is considered as “impolite” or even treated as denial-of-service (DOS) attack. For example, without any constraint, the crawler can send thousands of requests every second to the same website. This can overwhelm the web servers.

The general idea of enforcing politeness is to download one page at a time from the same host. A delay can be added between two download tasks. The politeness constraint is implemented by maintain a mapping from website hostnames to download (worker) threads. Each downloader thread has a separate FIFO queue and only downloads URLs obtained from that queue. Figure 9-6 shows the design that manages politeness.

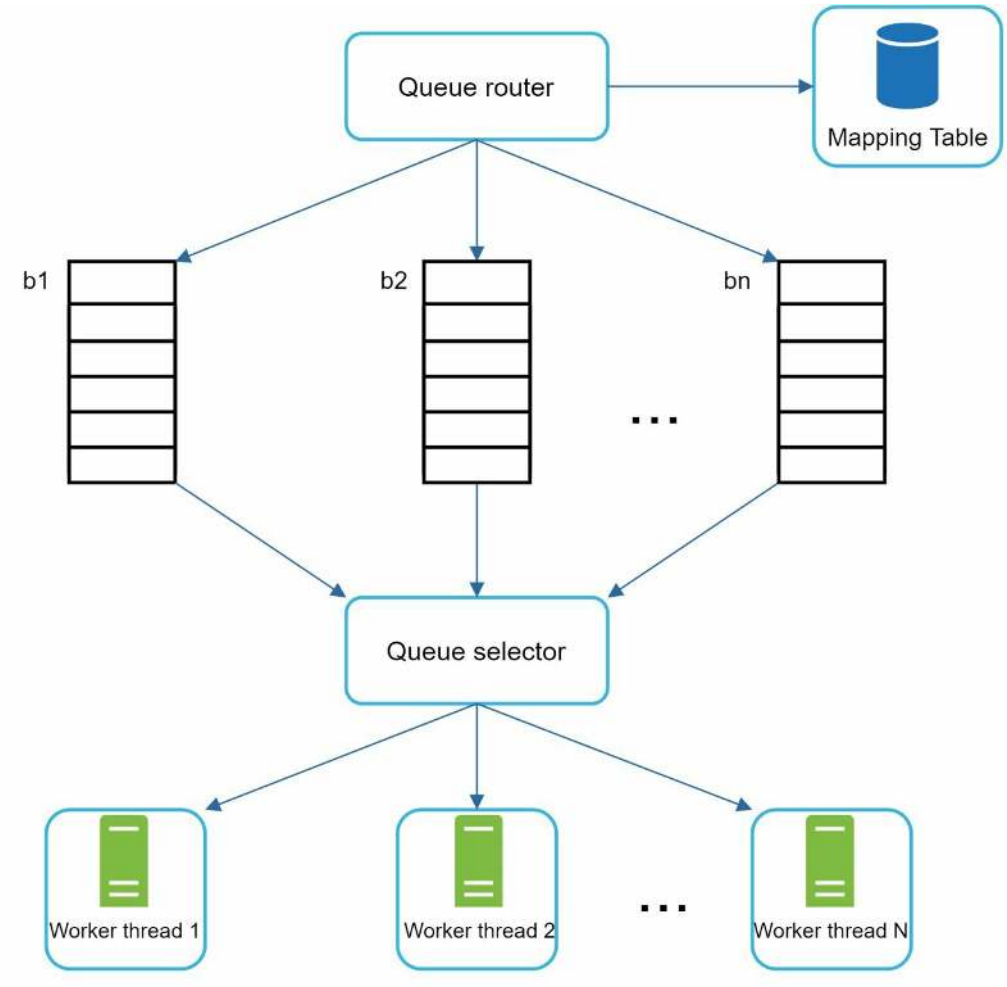


Figure 9-6

• Queue router: It ensures that each queue (b1, b2, … bn) only contains URLs from the same host.

• Mapping table: It maps each host to a queue.

Host	Queue
wikipedia.com	b1
apple.com	b2
...	...
nike.com	bn

Table 9-1

• FIFO queues b1, b2 to bn: Each queue contains URLs from the same host.

• Queue selector: Each worker thread is mapped to a FIFO queue, and it only downloads URLs from that queue. The queue selection logic is done by the Queue selector.

• Worker thread 1 to N. A worker thread downloads web pages one by one from the same host. A delay can be added between two download tasks.

Priority

A random post from a discussion forum about Apple products carries very different weight than posts on the Apple home page. Even though they both have the “Apple” keyword, it is sensible for a crawler to crawl the Apple home page first.

We prioritize URLs based on usefulness, which can be measured by PageRank [10], website traffic, update frequency, etc. “Prioritizer” is the component that handles URL prioritization. Refer to the reference materials [5] [10] for in-depth information about this concept.

Figure 9-7 shows the design that manages URL priority.

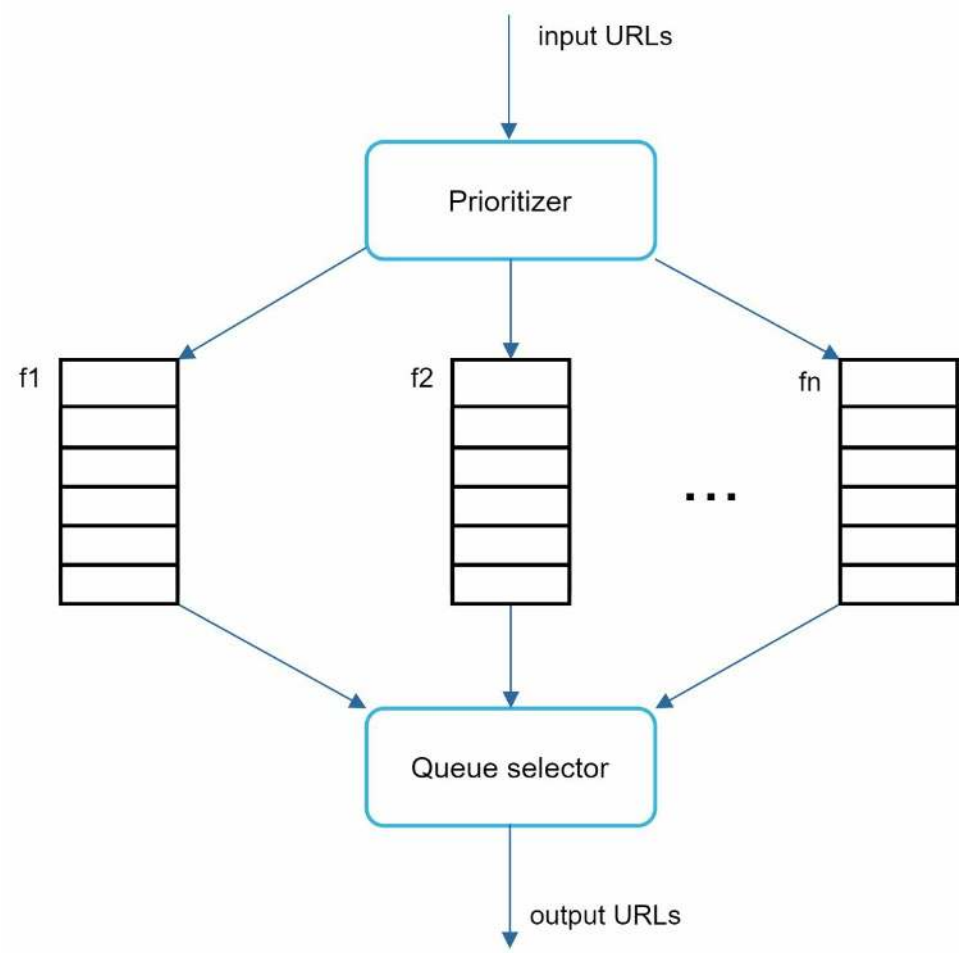


Figure 9-7

- Prioritizer: It takes URLs as input and computes the priorities.
- Queue f1 to fn: Each queue has an assigned priority. Queues with high priority are selected with higher probability.
- Queue selector: Randomly choose a queue with a bias towards queues with higher priority.

Figure 9-8 presents the URL frontier design, and it contains two modules:

- Front queues: manage prioritization
- Back queues: manage politeness

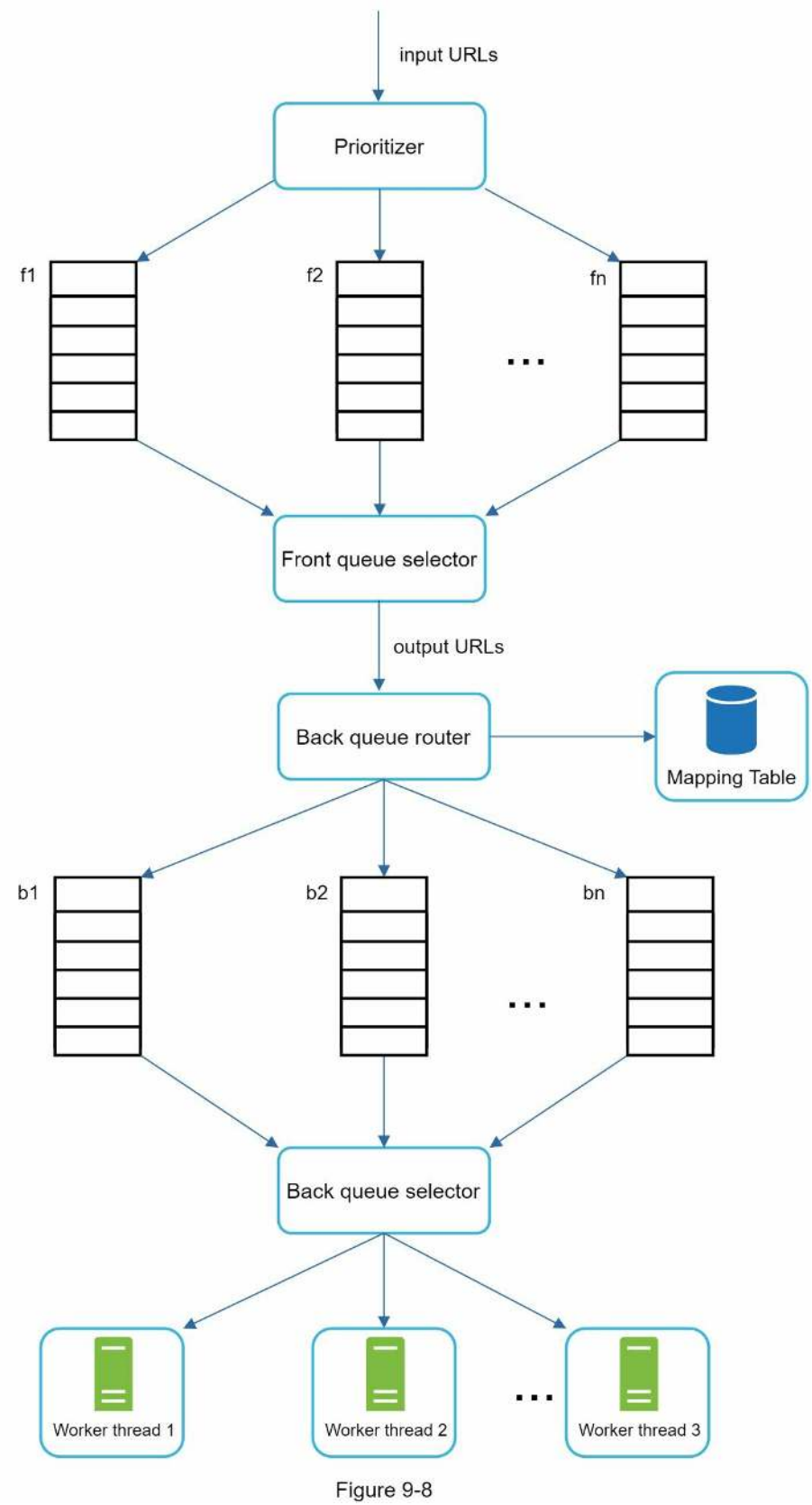


Figure 9-8

Freshness

Web pages are constantly being added, deleted, and edited. A web crawler must periodically recrawl downloaded pages to keep our data set fresh. Recrawl all the URLs is time-consuming and resource intensive. Few strategies to optimize freshness are listed as follows:

- Recrawl based on web pages' update history.
- Prioritize URLs and recrawl important pages first and more frequently.

Storage for URL Frontier

In real-world crawl for search engines, the number of URLs in the frontier could be hundreds of millions [4]. Putting everything in memory is neither durable nor scalable. Keeping everything in the disk is undesirable neither because the disk is slow; and it can easily become a bottleneck for the crawl.

We adopted a hybrid approach. The majority of URLs are stored on disk, so the storage space is not a problem. To reduce the cost of reading from the disk and writing to the disk, we maintain buffers in memory for enqueue/dequeue operations. Data in the buffer is periodically written to the disk.

HTML Downloader

The HTML Downloader downloads web pages from the internet using the HTTP protocol. Before discussing the HTML Downloader, we look at Robots Exclusion Protocol first.

Robots.txt

Robots.txt, called Robots Exclusion Protocol, is a standard used by websites to communicate with crawlers. It specifies what pages crawlers are allowed to download. Before attempting to crawl a web site, a crawler should check its corresponding robots.txt first and follow its rules.

To avoid repeat downloads of robots.txt file, we cache the results of the file. The file is downloaded and saved to cache periodically. Here is a piece of robots.txt file taken from https://www.amazon.com/robots.txt. Some of the directories like creatorhub are disallowed for Google bot.

```
User-agent: Googlebot
Disallow: /creatorhub/*
Disallow: /rss/people/*/reviews
Disallow: /gp/pdp/rss/*/reviews
Disallow: /gp/cdp/member-reviews/
Disallow: /gp/aw/cr/
```

Besides robots.txt, performance optimization is another important concept we will cover for the HTML downloader.

Performance optimization

Below is a list of performance optimizations for HTML downloader.

1. Distributed crawl

To achieve high performance, crawl jobs are distributed into multiple servers, and each server runs multiple threads. The URL space is partitioned into smaller pieces; so, each downloader is responsible for a subset of the URLs. Figure 9-9 shows an example of a distributed crawl.

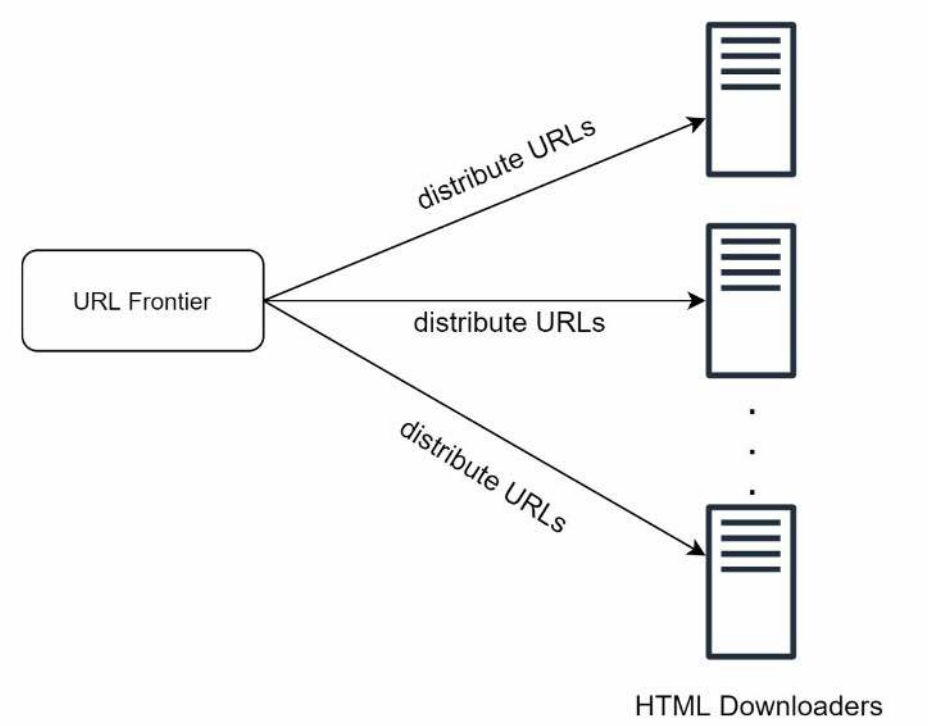


Figure 9-9

2. Cache DNS Resolver

DNS Resolver is a bottleneck for crawlers because DNS requests might take time due to the synchronous nature of many DNS interfaces. DNS response time ranges from 10ms to 200ms. Once a request to DNS is carried out by a crawler thread, other threads are blocked until the first request is completed. Maintaining our DNS cache to avoid calling DNS frequently is an effective technique for speed optimization. Our DNS cache keeps the domain name to IP address mapping and is updated periodically by cron jobs.

3. Locality

Distribute crawl servers geographically. When crawl servers are closer to website hosts, crawlers experience faster download time. Design locality applies to most of the system components: crawl servers, cache, queue, storage, etc.

4. Short timeout

Some web servers respond slowly or may not respond at all. To avoid long wait time, a maximal wait time is specified. If a host does not respond within a predefined time, the crawler will stop the job and crawl some other pages.

Robustness

Besides performance optimization, robustness is also an important consideration. We present a few approaches to improve the system robustness:

- Consistent hashing: This helps to distribute loads among downloaders. A new downloader server can be added or removed using consistent hashing. Refer to Chapter 5: Design consistent hashing for more details.
- Save crawl states and data: To guard against failures, crawl states and data are written to a storage system. A disrupted crawl can be restarted easily by loading saved states and data.
- Exception handling: Errors are inevitable and common in a large-scale system. The

crawler must handle exceptions gracefully without crashing the system.

• Data validation: This is an important measure to prevent system errors.

Extensibility

As almost every system evolves, one of the design goals is to make the system flexible enough to support new content types. The crawler can be extended by plugging in new modules. Figure 9-10 shows how to add new modules.

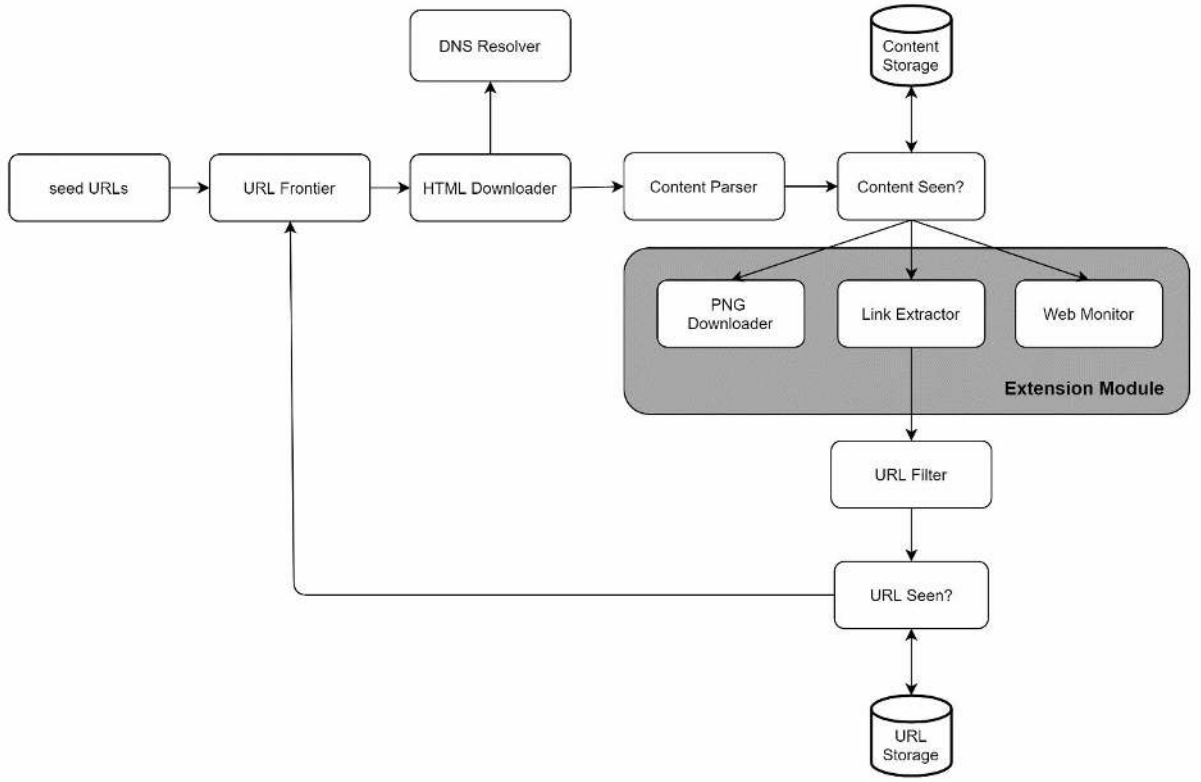


Figure 9-10

• PNG Downloader module is plugged-in to download PNG files.

• Web Monitor module is added to monitor the web and prevent copyright and trademark infringements.

Detect and avoid problematic content

This section discusses the detection and prevention of redundant, meaningless, or harmful content.

1. Redundant content

As discussed previously, nearly 30% of the web pages are duplicates. Hashes or checksums help to detect duplication [11].

2. Spider traps

A spider trap is a web page that causes a crawler in an infinite loop. For instance, an infinite deep directory structure is listed as follows:
www.spidertrapexample.com/foo/bar/foo/bar/foo/bar/…

Such spider traps can be avoided by setting a maximal length for URLs. However, no one-size-fits-all solution exists to detect spider traps. Websites containing spider traps are easy to identify due to an unusually large number of web pages discovered on such websites. It is hard to develop automatic algorithms to avoid spider traps; however, a user can manually

verify and identify a spider trap, and either exclude those websites from the crawler or apply some customized URL filters.

3. Data noise

Some of the contents have little or no value, such as advertisements, code snippets, spam URLs, etc. Those contents are not useful for crawlers and should be excluded if possible.

Step 4 - Wrap up

In this chapter, we first discussed the characteristics of a good crawler: scalability, politeness, extensibility, and robustness. Then, we proposed a design and discussed key components. Building a scalable web crawler is not a trivial task because the web is enormously large and full of traps. Even though we have covered many topics, we still miss many relevant talking points:

- Server-side rendering: Numerous websites use scripts like JavaScript, AJAX, etc to generate links on the fly. If we download and parse web pages directly, we will not be able to retrieve dynamically generated links. To solve this problem, we perform server-side rendering (also called dynamic rendering) first before parsing a page [12].
- Filter out unwanted pages: With finite storage capacity and crawl resources, an anti-spam component is beneficial in filtering out low quality and spam pages [13] [14].
- Database replication and sharding: Techniques like replication and sharding are used to improve the data layer availability, scalability, and reliability.
- Horizontal scaling: For large scale crawl, hundreds or even thousands of servers are needed to perform download tasks. The key is to keep servers stateless.
- Availability, consistency, and reliability: These concepts are at the core of any large system's success. We discussed these concepts in detail in Chapter 1. Refresh your memory on these topics.
- Analytics: Collecting and analyzing data are important parts of any system because data is key ingredient for fine-tuning.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Reference materials

[1] US Library of Congress: https://www.loc.gov/websites/

[2] EU Web Archive: http://data.europa.eu/webarchive

[3] Digimarc: https://www.digimarc.com/products/digimarc-services/piracy-intelligence

[4] Heydon A., Najork M. Mercator: A scalable, extensible web crawler World Wide Web, 2 (4) (1999), pp. 219-229

[5] By Christopher Olston, Marc Najork: Web Crawling. http://infolab.stanford.edu/~olston/publications/crawling_survey.pdf

[6] 29% Of Sites Face Duplicate Content Issues: https://tinyurl.com/y6tmh55y

[7] Rabin M.O., et al. Fingerprinting by random polynomials Center for Research in Computing Techn., Aiken Computation Laboratory, Univ. (1981)

[8] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” Communications of the ACM, vol. 13, no. 7, pp. 422–426, 1970.

[9] Donald J. Patterson, Web Crawling: https://www.ics.uci.edu/~lopes/teaching/cs221W12/slides/Lecture05.pdf

[10] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank citation ranking: Bringing order to the web,” Technical Report, Stanford University, 1998.

[11] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7), pages 422--426, July 1970.

[12] Google Dynamic Rendering: https://developers.google.com/search/docs/guides/dynamic-rendering

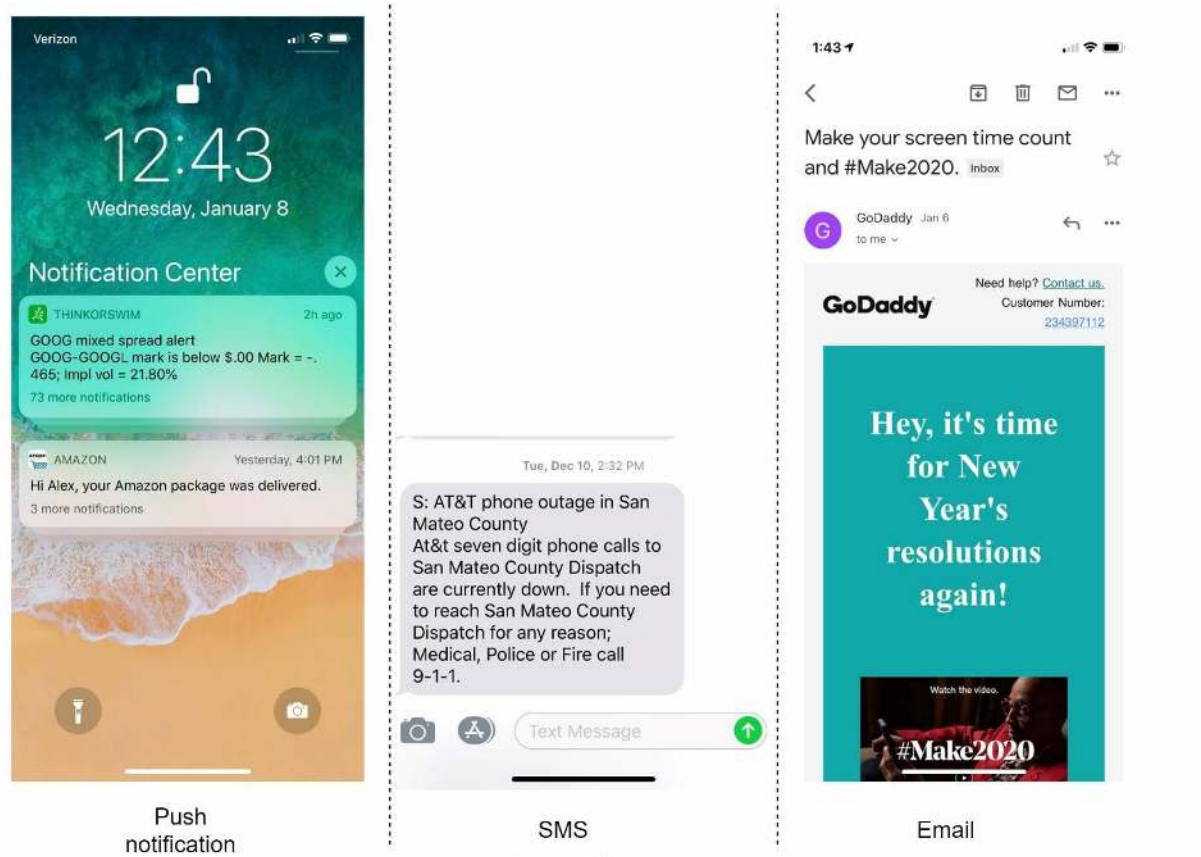
[13] T. Urvoy, T. Lavergne, and P. Filoche, “Tracking web spam with hidden style similarity,” in Proceedings of the 2nd International Workshop on Adversarial Information Retrieval on the Web, 2006.

[14] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov, “IRLbot: Scaling to 6 billion pages and beyond,” in Proceedings of the 17th International World Wide Web Conference, 2008.

CHAPTER 10: DESIGN A NOTIFICATION SYSTEM

A notification system has already become a very popular feature for many applications in recent years. A notification alerts a user with important information like breaking news, product updates, events, offerings, etc. It has become an indispensable part of our daily life. In this chapter, you are asked to design a notification system.

A notification is more than just mobile push notification. Three types of notification formats are: mobile push notification, SMS message, and Email. Figure 10-1 shows an example of each of these notifications.



Push notification

SMS

Email

Figure 10-1

Step 1 - Understand the problem and establish design scope

Building a scalable system that sends out millions of notifications a day is not an easy task. It requires a deep understanding of the notification ecosystem. The interview question is purposely designed to be open-ended and ambiguous, and it is your responsibility to ask questions to clarify the requirements.

Candidate: What types of notifications does the system support?
Interviewer: Push notification, SMS message, and email.

Candidate: Is it a real-time system?
Interviewer: Let us say it is a soft real-time system. We want a user to receive notifications as soon as possible. However, if the system is under a high workload, a slight delay is acceptable.

Candidate: What are the supported devices?
Interviewer: iOS devices, android devices, and laptop/desktop.

Candidate: What triggers notifications?
Interviewer: Notifications can be triggered by client applications. They can also be scheduled on the server-side.

Candidate: Will users be able to opt-out?
Interviewer: Yes, users who choose to opt-out will no longer receive notifications.

Candidate: How many notifications are sent out each day?
Interviewer: 10 million mobile push notifications, 1 million SMS messages, and 5 million emails.

Step 2 - Propose high-level design and get buy-in

This section shows the high-level design that supports various notification types: iOS push notification, Android push notification, SMS message, and Email. It is structured as follows:

- Different types of notifications
- Contact info gathering flow
- Notification sending/receiving flow

Different types of notifications

We start by looking at how each notification type works at a high level.

iOS push notification

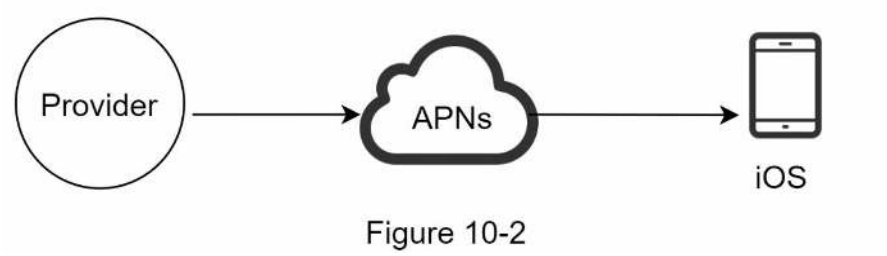


Figure 10-2

We primary need three components to send an iOS push notification:

- Provider. A provider builds and sends notification requests to Apple Push Notification Service (APNS). To construct a push notification, the provider provides the following data:
  - Device token: This is a unique identifier used for sending push notifications.
  - Payload: This is a JSON dictionary that contains a notification's payload. Here is an example:

```
{
  "aps" : {
              "alert" : {
                 "title" : "Game Request",
                 "body" : "Bob wants to play chess",
                 "action-loc-key" : "PLAY"
              },
              "badge" : 5
           }
}
```

- APNS: This is a remote service provided by Apple to propagate push notifications to iOS devices.
- iOS Device: It is the end client, which receives push notifications.

Android push notification

Android adopts a similar notification flow. Instead of using APNs, Firebase Cloud Messaging (FCM) is commonly used to send push notifications to android devices.

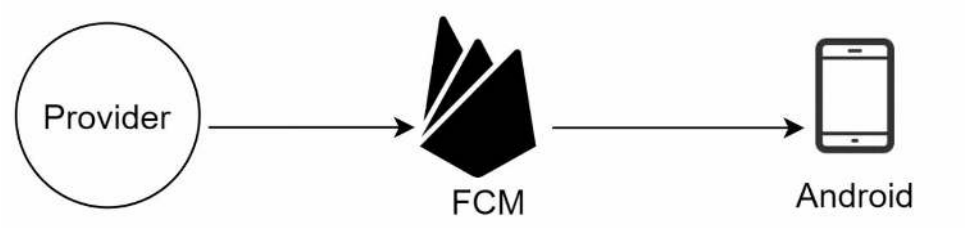


Figure 10-3

SMS message

For SMS messages, third party SMS services like Twilio [1], Nexmo [2], and many others are commonly used. Most of them are commercial services.

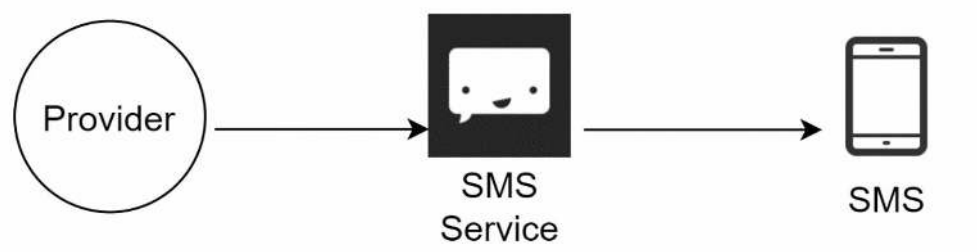


Figure 10-4

Email

Although companies can set up their own email servers, many of them opt for commercial email services. Sendgrid [3] and Mailchimp [4] are among the most popular email services, which offer a better delivery rate and data analytics.

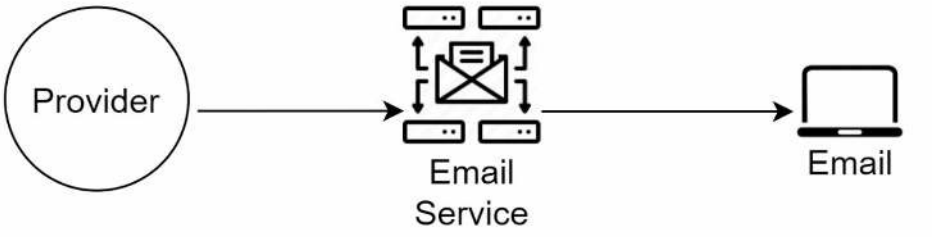


Figure 10-5

Figure 10-6 shows the design after including all the third-party services.

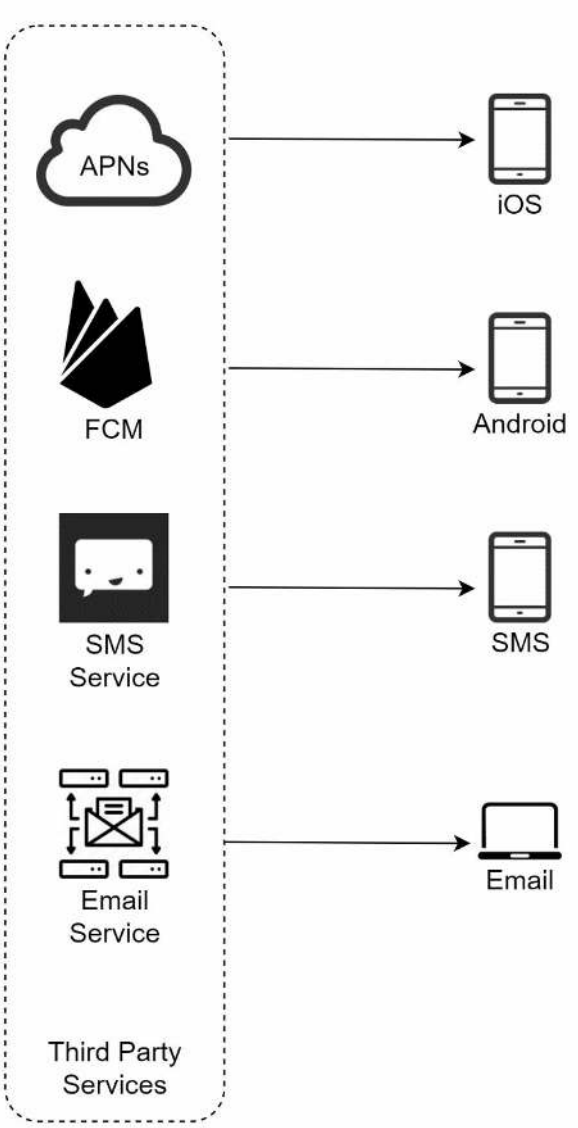


Figure 10-6

Contact info gathering flow

To send notifications, we need to gather mobile device tokens, phone numbers, or email addresses. As shown in Figure 10-7, when a user installs our app or signs up for the first time, API servers collect user contact info and store it in the database.

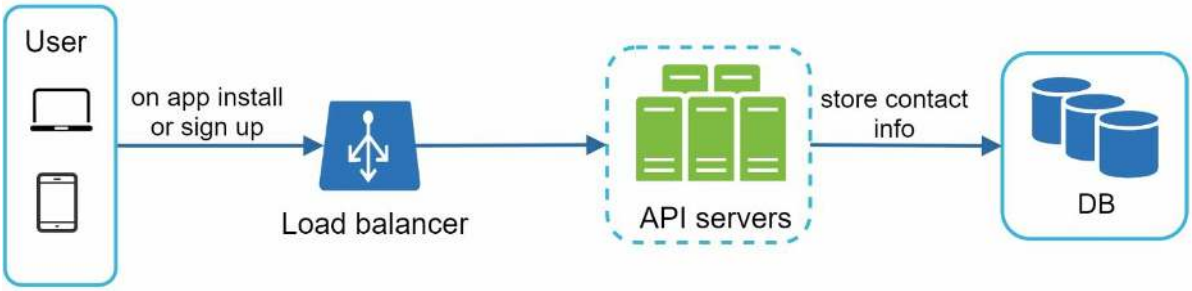


Figure 10-7

Figure 10-8 shows simplified database tables to store contact info. Email addresses and phone numbers are stored in the *user* table, whereas device tokens are stored in the *device* table. A

user can have multiple devices, indicating that a push notification can be sent to all the user devices.

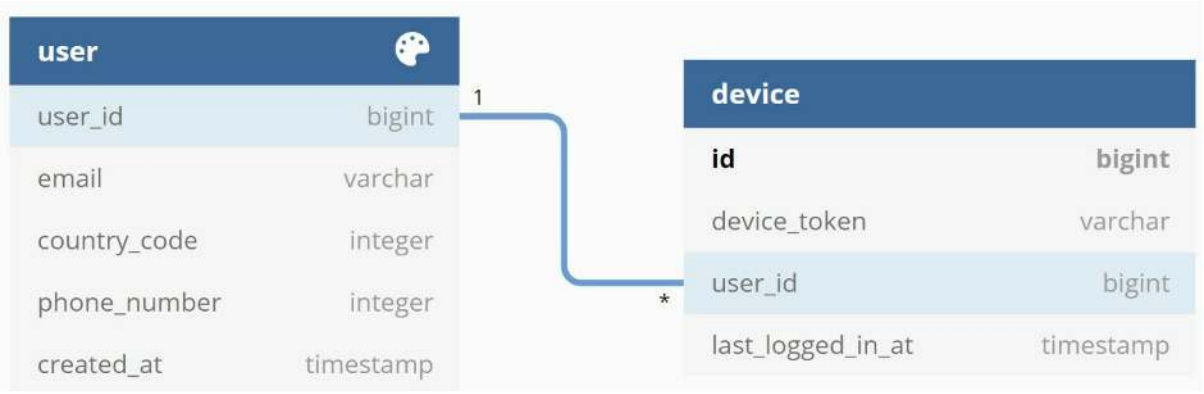


Figure 10-8

Notification sending/receiving flow

We will first present the initial design; then, propose some optimizations.

High-level design

Figure 10-9 shows the design, and each system component is explained below.

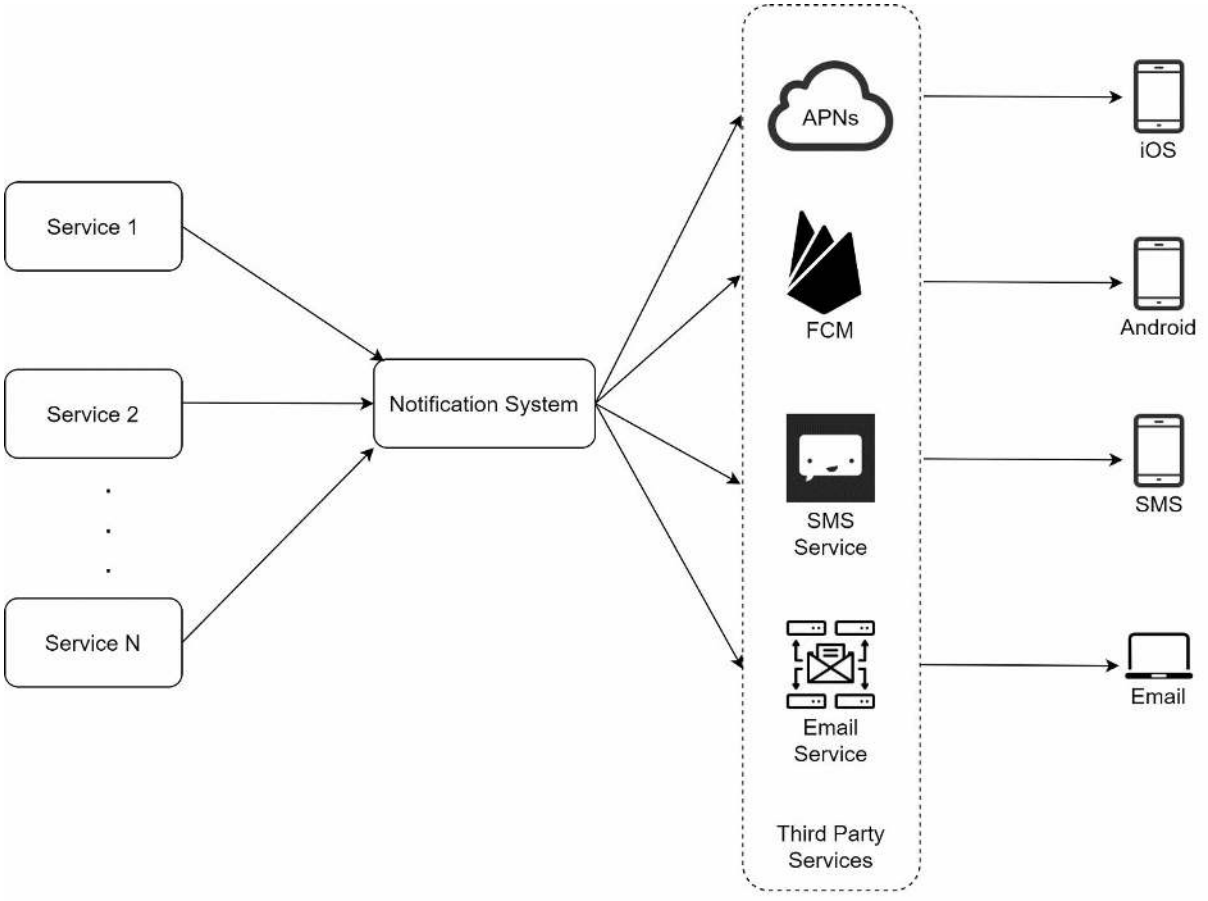


Figure 10-9

Service 1 to N: A service can be a micro-service, a cron job, or a distributed system that triggers notification sending events. For example, a billing service sends emails to remind customers of their due payment or a shopping website tells customers that their packages will be delivered tomorrow via SMS messages.

Notification system: The notification system is the centerpiece of sending/receiving notifications. Starting with something simple, only one notification server is used. It provides APIs for services 1 to N, and builds notification payloads for third party services.

Third-party services: Third party services are responsible for delivering notifications to users. While integrating with third-party services, we need to pay extra attention to extensibility. Good extensibility means a flexible system that can easily plugging or unplugging of a third-party service. Another important consideration is that a third-party service might be unavailable in new markets or in the future. For instance, FCM is unavailable in China. Thus, alternative third-party services such as Jpush, PushY, etc are used there.

iOS, Android, SMS, Email: Users receive notifications on their devices.

Three problems are identified in this design:

- Single point of failure (SPOF): A single notification server means SPOF.
- Hard to scale: The notification system handles everything related to push notifications in one server. It is challenging to scale databases, caches, and different notification processing components independently.
- Performance bottleneck: Processing and sending notifications can be resource intensive. For example, constructing HTML pages and waiting for responses from third party services could take time. Handling everything in one system can result in the system overload, especially during peak hours.

High-level design (improved)

After enumerating challenges in the initial design, we improve the design as listed below:

- Move the database and cache out of the notification server.
- Add more notification servers and set up automatic horizontal scaling.
- Introduce message queues to decouple the system components.

Figure 10-10 shows the improved high-level design.

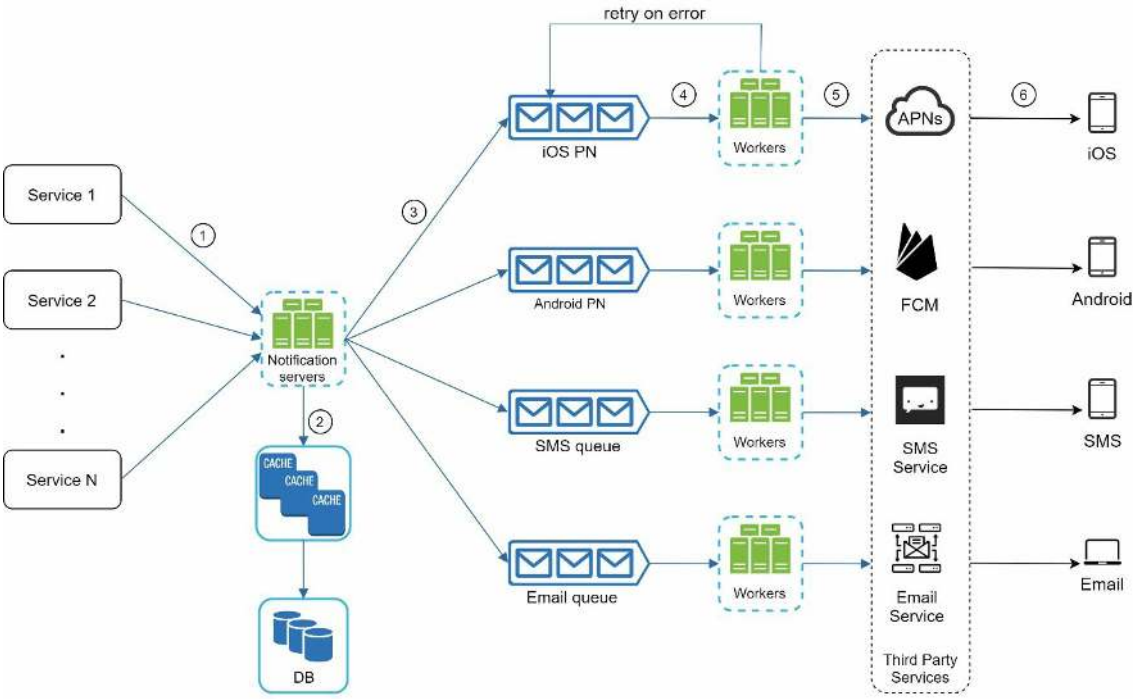


Figure 10-10

The best way to go through the above diagram is from left to right:

Service 1 to N: They represent different services that send notifications via APIs provided by notification servers.

Notification servers: They provide the following functionalities:

- Provide APIs for services to send notifications. Those APIs are only accessible internally or by verified clients to prevent spams.
- Carry out basic validations to verify emails, phone numbers, etc.
- Query the database or cache to fetch data needed to render a notification.
- Put notification data to message queues for parallel processing.

Here is an example of the API to send an email:

POST https://api.example.com/v/sms/send

Request body

```
{
  "to": [
    {
      "user_id": 123456
    }
  ],
  "from": {
    "email": "from_address@example.com"
  },
  "subject": "Hello, World!",
  "content": [
    {
      "type": "text/plain",
      "value": "Hello, World!"
    }
  ]
}
```

Cache: User info, device info, notification templates are cached.

DB: It stores data about user, notification, settings, etc.

Message queues: They remove dependencies between components. Message queues serve as buffers when high volumes of notifications are to be sent out. Each notification type is assigned with a distinct message queue so an outage in one third-party service will not affect other notification types.

Workers: Workers are a list of servers that pull notification events from message queues and send them to the corresponding third-party services.

Third-party services: Already explained in the initial design.

iOS, Android, SMS, Email: Already explained in the initial design.

Next, let us examine how every component works together to send a notification:

1. A service calls APIs provided by notification servers to send notifications.
2. Notification servers fetch metadata such as user info, device token, and notification setting from the cache or database.
3. A notification event is sent to the corresponding queue for processing. For instance, an iOS push notification event is sent to the iOS PN queue.
4. Workers pull notification events from message queues.
5. Workers send notifications to third party services.
6. Third-party services send notifications to user devices.

Step 3 - Design deep dive

In the high-level design, we discussed different types of notifications, contact info gathering flow, and notification sending/receiving flow. We will explore the following in deep dive:

- Reliability.
- Additional component and considerations: notification template, notification settings, rate limiting, retry mechanism, security in push notifications, monitor queued notifications and event tracking.
- Updated design.

Reliability

We must answer a few important reliability questions when designing a notification system in distributed environments.

How to prevent data loss?

One of the most important requirements in a notification system is that it cannot lose data. Notifications can usually be delayed or re-ordered, but never lost. To satisfy this requirement, the notification system persists notification data in a database and implements a retry mechanism. The notification log database is included for data persistence, as shown in Figure 10-11.

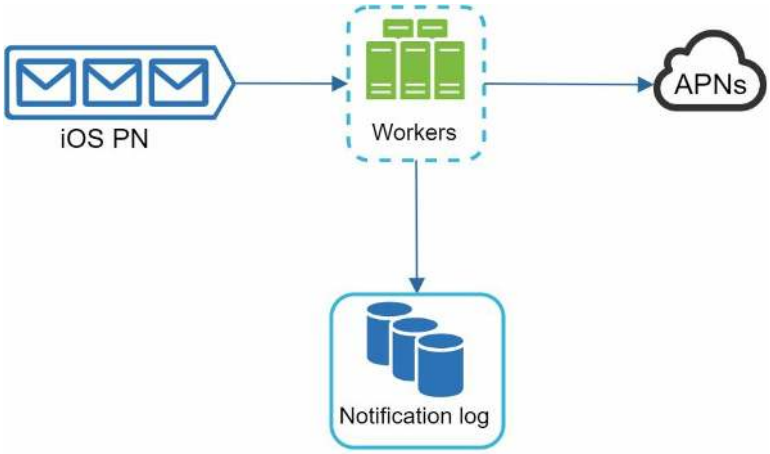


Figure 10-11

Will recipients receive a notification exactly once?

The short answer is no. Although notification is delivered exactly once most of the time, the distributed nature could result in duplicate notifications. To reduce the duplication occurrence, we introduce a dedupe mechanism and handle each failure case carefully. Here is a simple dedupe logic:

When a notification event first arrives, we check if it is seen before by checking the event ID. If it is seen before, it is discarded. Otherwise, we will send out the notification. For interested readers to explore why we cannot have exactly once delivery, refer to the reference material [5].

Additional components and considerations

We have discussed how to collect user contact info, send, and receive a notification. A notification system is a lot more than that. Here we discuss additional components including template reusing, notification settings, event tracking, system monitoring, rate limiting, etc.

Notification template