A large notification system sends out millions of notifications per day, and many of these notifications follow a similar format. Notification templates are introduced to avoid building every notification from scratch. A notification template is a preformatted notification to create your unique notification by customizing parameters, styling, tracking links, etc. Here is an example template of push notifications.

BODY:

You dreamed of it. We dared it. [ITEM NAME] is back — only until [DATE].

CTA:

Order Now. Or, Save My [ITEM NAME]

The benefits of using notification templates include maintaining a consistent format, reducing the margin error, and saving time.

Notification setting

Users generally receive way too many notifications daily and they can easily feel overwhelmed. Thus, many websites and apps give users fine-grained control over notification settings. This information is stored in the notification setting table, with the following fields:

```
user_id   bigInt
channel  varchar    # push notification, email or SMS
opt_in   boolean    # opt-in to receive notification
```

Before any notification is sent to a user, we first check if a user is opted-in to receive this type of notification.

Rate limiting

To avoid overwhelming users with too many notifications, we can limit the number of notifications a user can receive. This is important because receivers could turn off notifications completely if we send too often.

Retry mechanism

When a third-party service fails to send a notification, the notification will be added to the message queue for retrying. If the problem persists, an alert will be sent out to developers.

Security in push notifications

For iOS or Android apps, appKey and appSecret are used to secure push notification APIs [6]. Only authenticated or verified clients are allowed to send push notifications using our APIs. Interested users should refer to the reference material [6].

Monitor queued notifications

A key metric to monitor is the total number of queued notifications. If the number is large, the notification events are not processed fast enough by workers. To avoid delay in the notification delivery, more workers are needed. Figure 10-12 (credit to [7]) shows an example of queued messages to be processed.

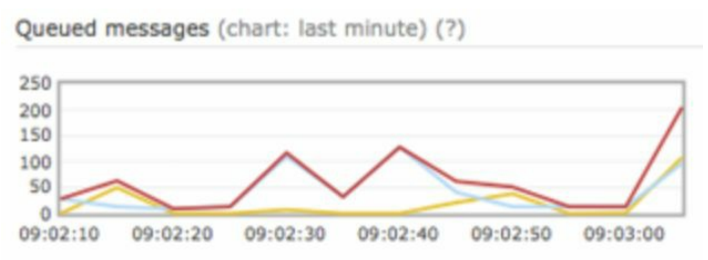

Figure 10-12

Events tracking

Notification metrics, such as open rate, click rate, and engagement are important in understanding customer behaviors. Analytics service implements events tracking. Integration between the notification system and the analytics service is usually required. Figure 10-13 shows an example of events that might be tracked for analytics purposes.

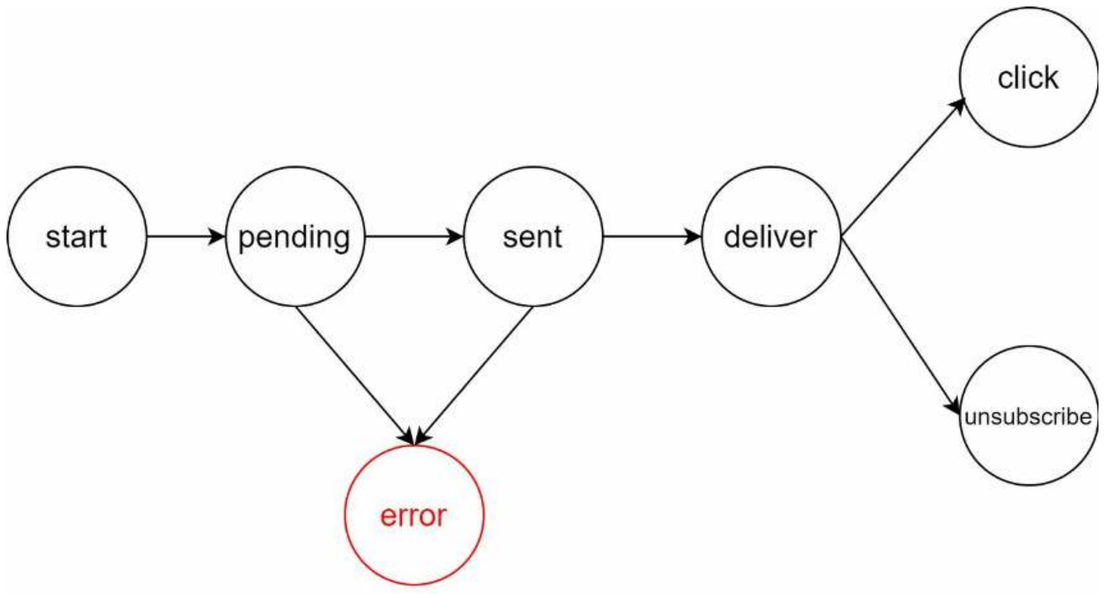

Figure 10-13

Updated design

Putting everything together, Figure 10-14 shows the updated notification system design.

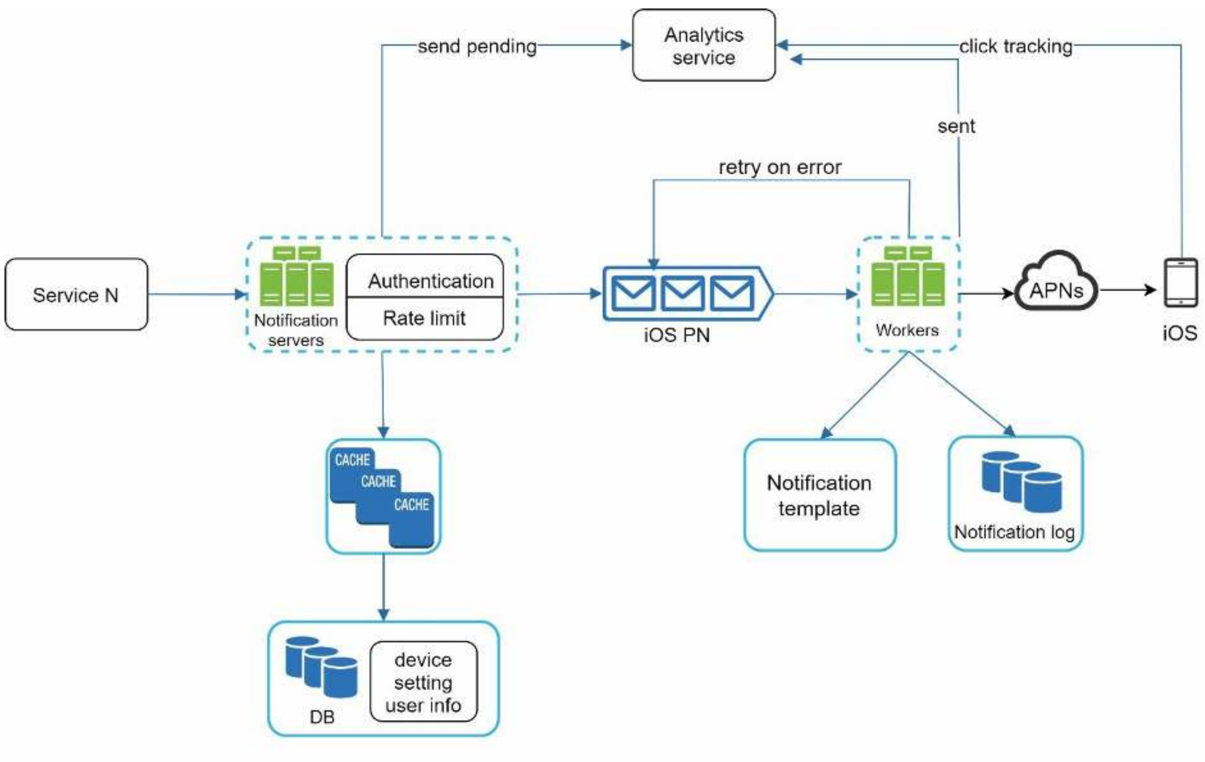

Figure 10-14

In this design, many new components are added in comparison with the previous design.

• The notification servers are equipped with two more critical features: authentication and rate-limiting.

• We also add a retry mechanism to handle notification failures. If the system fails to send notifications, they are put back in the messaging queue and the workers will retry for a predefined number of times.

• Furthermore, notification templates provide a consistent and efficient notification creation process.

• Finally, monitoring and tracking systems are added for system health checks and future improvements.

Step 4 - Wrap up

Notifications are indispensable because they keep us posted with important information. It could be a push notification about your favorite movie on Netflix, an email about discounts on new products, or a message about your online shopping payment confirmation.

In this chapter, we described the design of a scalable notification system that supports multiple notification formats: push notification, SMS message, and email. We adopted message queues to decouple system components.

Besides the high-level design, we dug deep into more components and optimizations.

- Reliability: We proposed a robust retry mechanism to minimize the failure rate.
- Security: AppKey/appSecret pair is used to ensure only verified clients can send notifications.
- Tracking and monitoring: These are implemented in any stage of a notification flow to capture important stats.
- Respect user settings: Users may opt-out of receiving notifications. Our system checks user settings first before sending notifications.
- Rate limiting: Users will appreciate a frequency capping on the number of notifications they receive.

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

Reference materials

[1] Twilio SMS: https://www.twilio.com/sms

[2] Nexmo SMS: https://www.nexmo.com/products/sms

[3] Sendgrid: https://sendgrid.com/

[4] Mailchimp: https://mailchimp.com/

[5] You Cannot Have Exactly-Once Delivery: https://bravenewgeek.com/you-cannot-have-exactly-once-delivery/

[6] Security in Push Notifications: https://cloud.ibm.com/docs/services/mobilepush?topic=mobile-pushnotification-security-in-push-notifications

[7] RadditMQ: https://bit.ly/2sotIa6

CHAPTER 11: DESIGN A NEWS FEED SYSTEM

In this chapter, you are asked to design a news feed system. What is news feed? According to the Facebook help page, “News feed is the constantly updating list of stories in the middle of your home page. News Feed includes status updates, photos, videos, links, app activity, and likes from people, pages, and groups that you follow on Facebook” [1]. This is a popular interview question. Similar questions commonly asked are: design Facebook news feed, Instagram feed, Twitter timeline, etc.

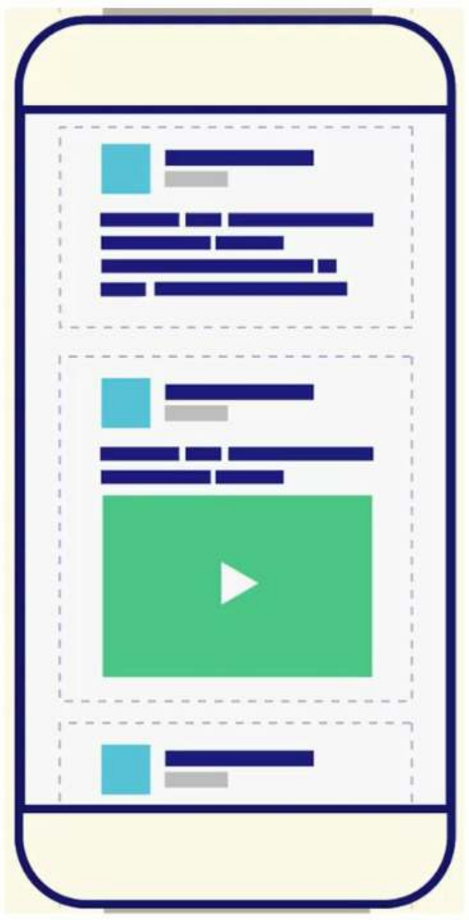

Figure 11-1 (source : https://bit.ly/2Vv9JCm)

Step 1 - Understand the problem and establish design scope

The first set of clarification questions are to understand what the interviewer has in mind when she asks you to design a news feed system. At the very least, you should figure out what features to support. Here is an example of candidate-interviewer interaction:

Candidate: Is this a mobile app? Or a web app? Or both?
Interviewer: Both

Candidate: What are the important features?
Interview: A user can publish a post and see her friends' posts on the news feed page.

Candidate: Is the news feed sorted by reverse chronological order or any particular order such as topic scores? For instance, posts from your close friends have higher scores.
Interviewer: To keep things simple, let us assume the feed is sorted by reverse chronological order.

Candidate: How many friends can a user have?
Interviewer: 5000

Candidate: What is the traffic volume?
Interviewer: 10 million DAU

Candidate: Can feed contain images, videos, or just text?
Interviewer: It can contain media files, including both images and videos.

Now you have gathered the requirements, we focus on designing the system.

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

The design is divided into two flows: feed publishing and news feed building.

- Feed publishing: when a user publishes a post, corresponding data is written into cache and database. A post is populated to her friends' news feed.
- Newsfeed building: for simplicity, let us assume the news feed is built by aggregating friends' posts in reverse chronological order.

Newsfeed APIs

The news feed APIs are the primary ways for clients to communicate with servers. Those APIs are HTTP based that allow clients to perform actions, which include posting a status, retrieving news feed, adding friends, etc. We discuss two most important APIs: feed publishing API and news feed retrieval API.

Feed publishing API

To publish a post, a HTTP POST request will be sent to the server. The API is shown below:

POST /v1/me/feed
Params:

- content: content is the text of the post.
- auth_token: it is used to authenticate API requests.

Newsfeed retrieval API

The API to retrieve news feed is shown below:

GET /v1/me/feed
Params:

- auth_token: it is used to authenticate API requests.

Feed publishing

Figure 11-2 shows the high-level design of the feed publishing flow.

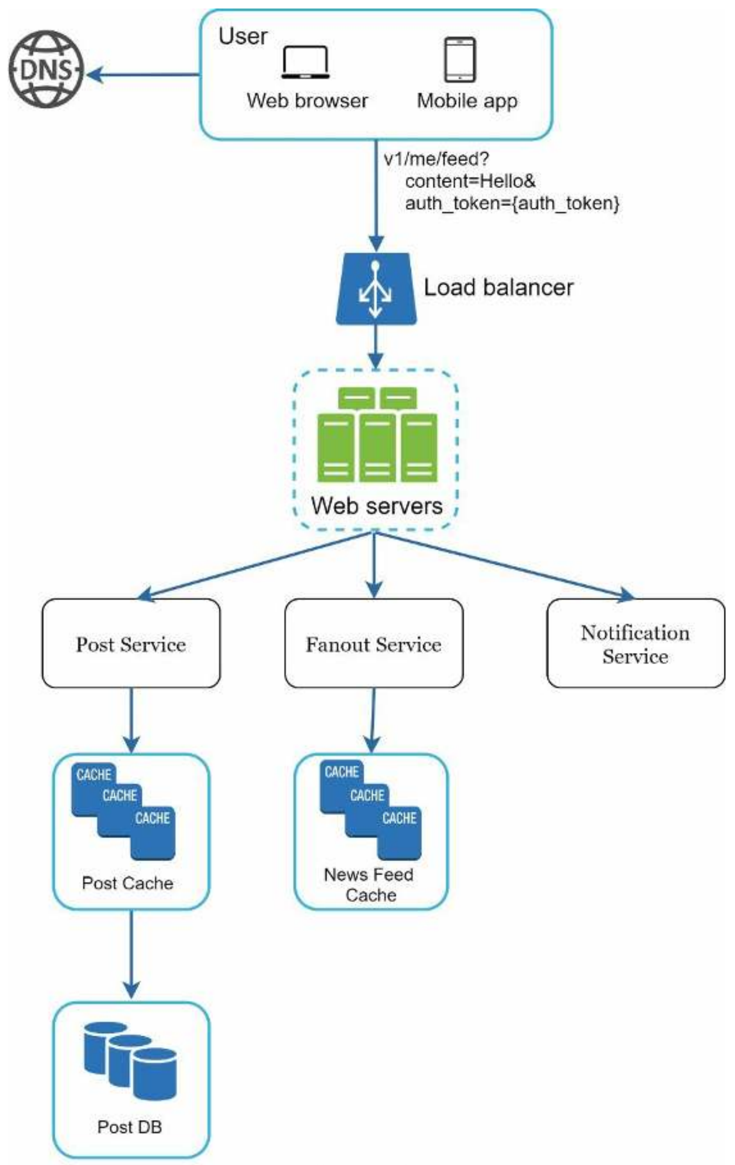

Figure 11-2

• User: a user can view news feeds on a browser or mobile app. A user makes a post with content "Hello" through API:

/v1/me/feed?content=Hello&auth_token={auth_token}

• Load balancer: distribute traffic to web servers.

• Web servers: web servers redirect traffic to different internal services.

• Post service: persist post in the database and cache.

• Fanout service: push new content to friends' news feed. Newsfeed data is stored in the cache for fast retrieval.

• Notification service: inform friends that new content is available and send out push notifications.

Newsfeed building

In this section, we discuss how news feed is built behind the scenes. Figure 11-3 shows the high-level design:

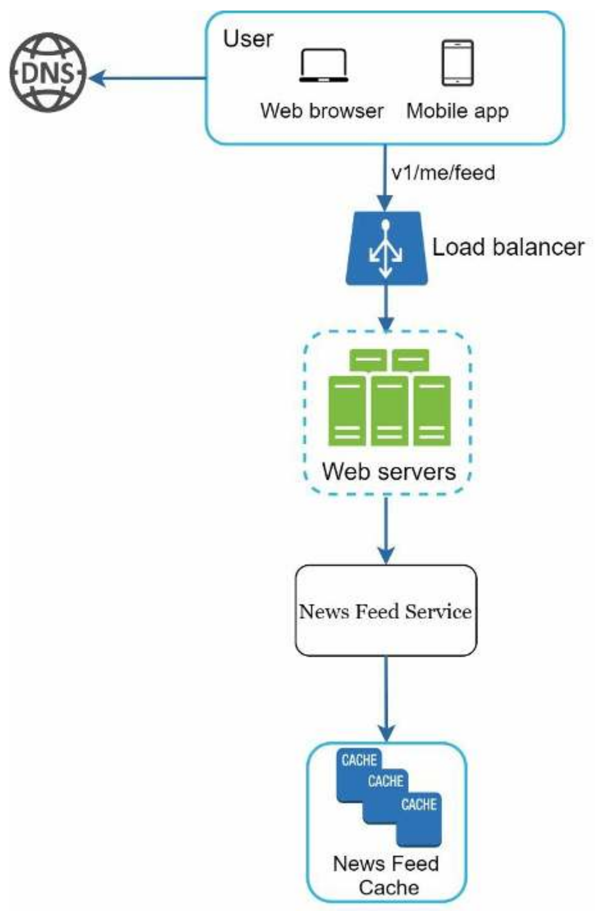

Figure 11-3

- User: a user sends a request to retrieve her news feed. The request looks like this: / v1/me/feed.
- Load balancer: load balancer redirects traffic to web servers.
- Web servers: web servers route requests to newsfeed service.
- Newsfeed service: news feed service fetches news feed from the cache.
- Newsfeed cache: store news feed IDs needed to render the news feed.

Step 3 - Design deep dive

The high-level design briefly covered two flows: feed publishing and news feed building. Here, we discuss those topics in more depth.

Feed publishing deep dive

Figure 11-4 outlines the detailed design for feed publishing. We have discussed most of components in high-level design, and we will focus on two components: web servers and fanout service.

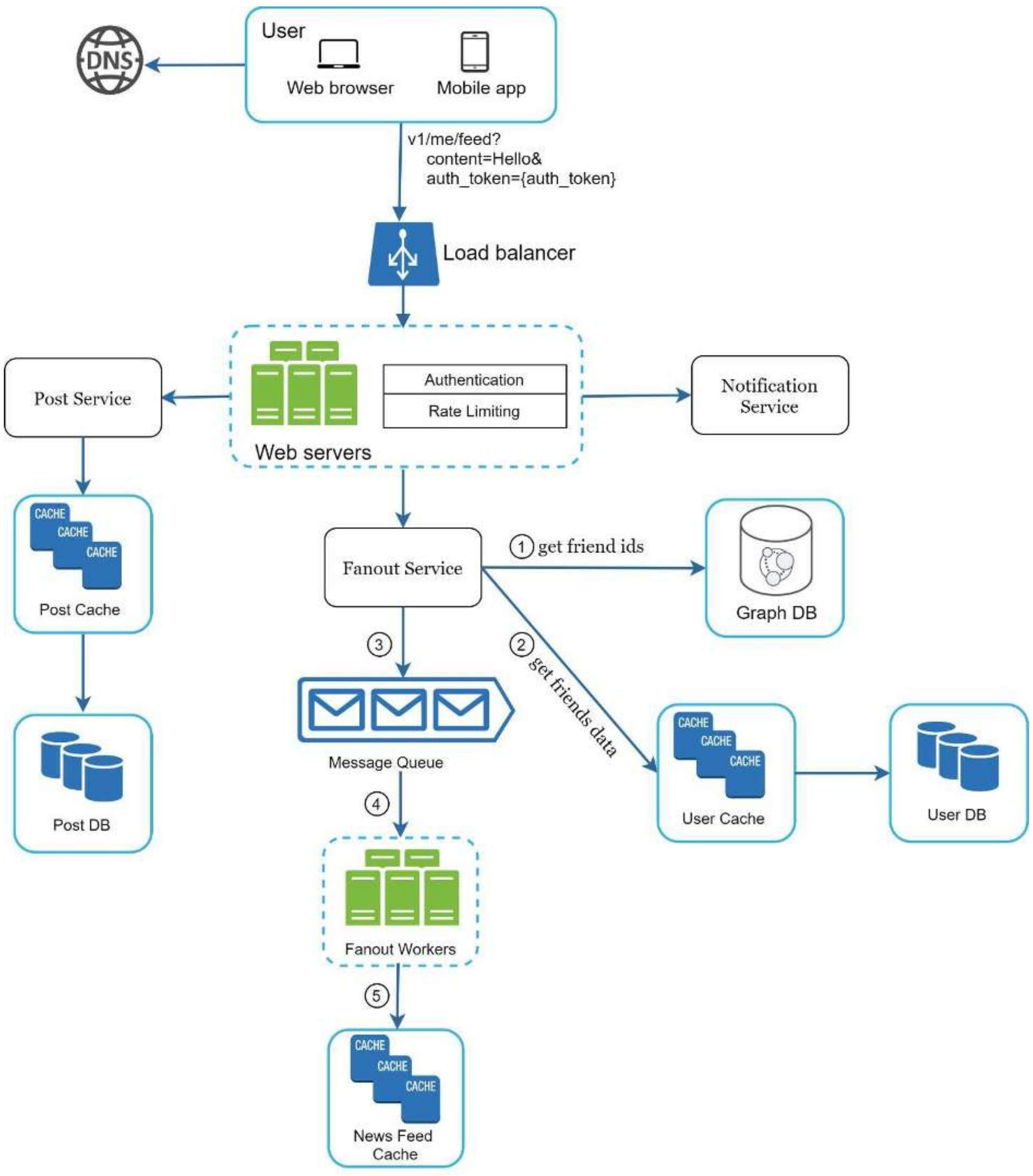

Figure 11-4

Web servers

Besides communicating with clients, web servers enforce authentication and rate-limiting.

Only users signed in with valid *auth_token* are allowed to make posts. The system limits the number of posts a user can make within a certain period, vital to prevent spam and abusive content.

Fanout service

Fanout is the process of delivering a post to all friends. Two types of fanout models are: fanout on write (also called push model) and fanout on read (also called pull model). Both models have pros and cons. We explain their workflows and explore the best approach to support our system.

Fanout on write. With this approach, news feed is pre-computed during write time. A new post is delivered to friends' cache immediately after it is published.

Pros:

- The news feed is generated in real-time and can be pushed to friends immediately.
- Fetching news feed is fast because the news feed is pre-computed during write time.

Cons:

- If a user has many friends, fetching the friend list and generating news feeds for all of them are slow and time consuming. It is called hotkey problem.
- For inactive users or those rarely log in, pre-computing news feeds waste computing resources.

Fanout on read. The news feed is generated during read time. This is an on-demand model. Recent posts are pulled when a user loads her home page.

Pros:

- For inactive users or those who rarely log in, fanout on read works better because it will not waste computing resources on them.
- Data is not pushed to friends so there is no hotkey problem.

Cons:

- Fetching the news feed is slow as the news feed is not pre-computed.

We adopt a hybrid approach to get benefits of both approaches and avoid pitfalls in them. Since fetching the news feed fast is crucial, we use a push model for the majority of users. For celebrities or users who have many friends/followers, we let followers pull news content on-demand to avoid system overload. Consistent hashing is a useful technique to mitigate the hotkey problem as it helps to distribute requests/data more evenly.

Let us take a close look at the fanout service as shown in Figure 11-5.

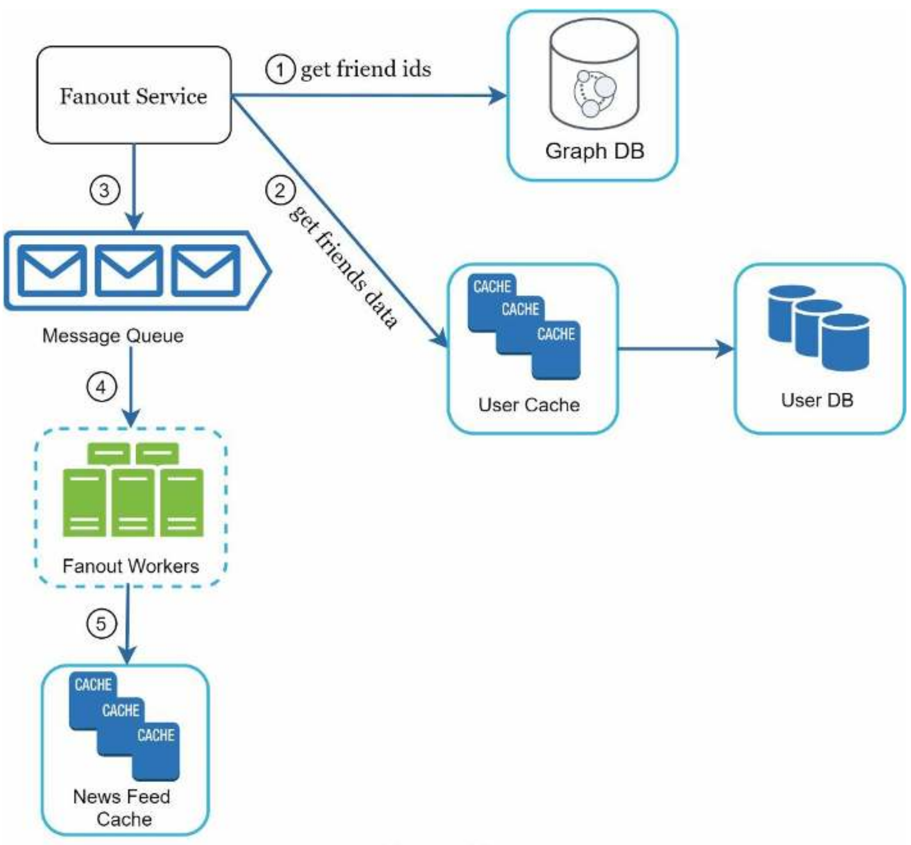

Figure 11-5

The fanout service works as follows:

1. Fetch friend IDs from the graph database. Graph databases are suited for managing friend relationship and friend recommendations. Interested readers wishing to learn more about this concept should refer to the reference material [2].

2. Get friends info from the user cache. The system then filters out friends based on user settings. For example, if you mute someone, her posts will not show up on your news feed even though you are still friends. Another reason why posts may not show is that a user could selectively share information with specific friends or hide it from other people.

3. Send friends list and new post ID to the message queue.

4. Fanout workers fetch data from the message queue and store news feed data in the news feed cache. You can think of the news feed cache as a <*post_id*, *user_id*> mapping table. Whenever a new post is made, it will be appended to the news feed table as shown in Figure 11-6. The memory consumption can become very large if we store the entire user and post objects in the cache. Thus, only IDs are stored. To keep the memory size small, we set a configurable limit. The chance of a user scrolling through thousands of posts in news feed is slim. Most users are only interested in the latest content, so the cache miss rate is low.

5. Store <*post_id*, *user_id* > in news feed cache. Figure 11-6 shows an example of what the news feed looks like in cache.

post_id	user_id
post_id	user_id
post_id	user_id
post_id	user_id
post_id	user_id
post_id	user_id
post_id	user_id
post_id	user_id

Figure 11-6

Newsfeed retrieval deep dive

Figure 11-7 illustrates the detailed design for news feed retrieval.

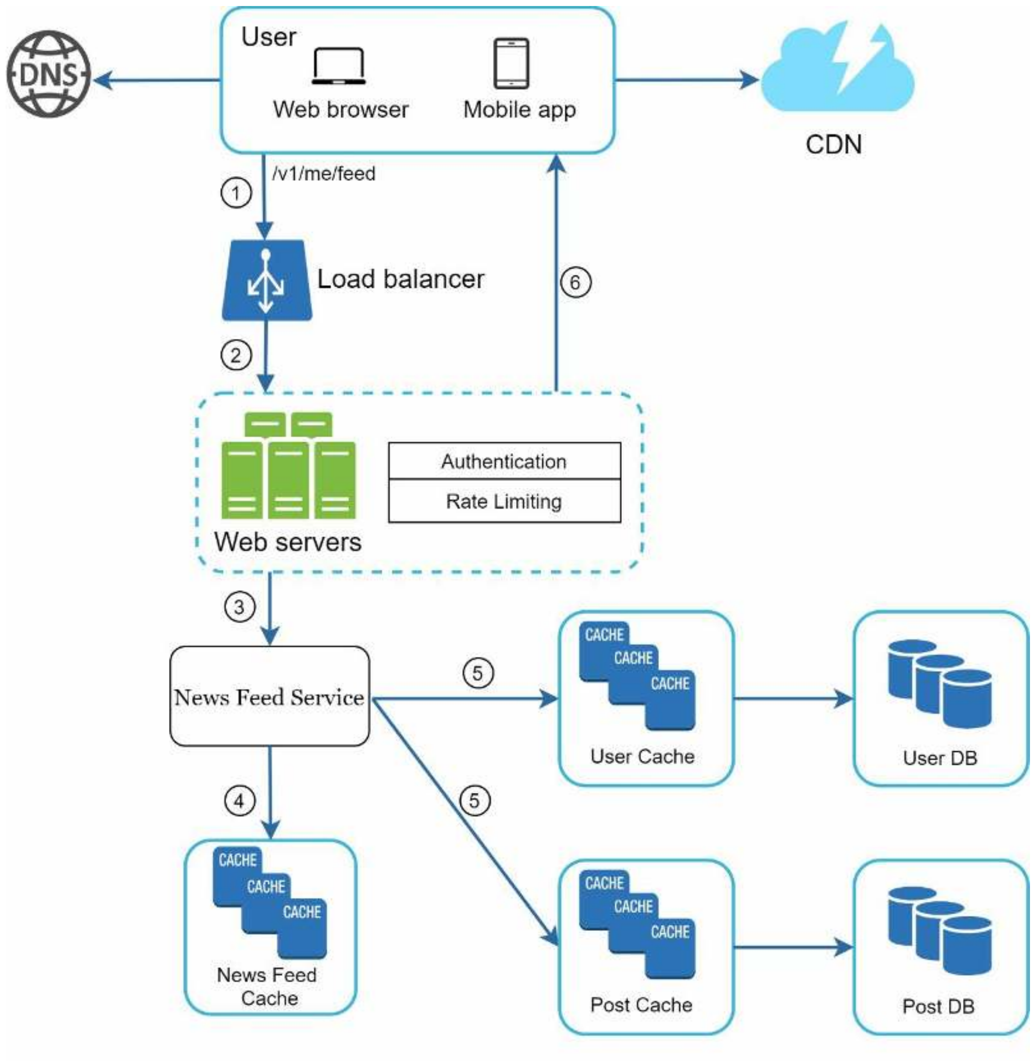

Figure 11-7

As shown in Figure 11-7, media content (images, videos, etc.) are stored in CDN for fast retrieval. Let us look at how a client retrieves news feed.

1. A user sends a request to retrieve her news feed. The request looks like this: */v1/me/feed*
2. The load balancer redistributes requests to web servers.
3. Web servers call the news feed service to fetch news feeds.
4. News feed service gets a list post IDs from the news feed cache.
5. A user’s news feed is more than just a list of feed IDs. It contains username, profile picture, post content, post image, etc. Thus, the news feed service fetches the complete user and post objects from caches (user cache and post cache) to construct the fully hydrated news feed.
6. The fully hydrated news feed is returned in JSON format back to the client for rendering.

Cache architecture

Cache is extremely important for a news feed system. We divide the cache tier into 5 layers as shown in Figure 11-8.

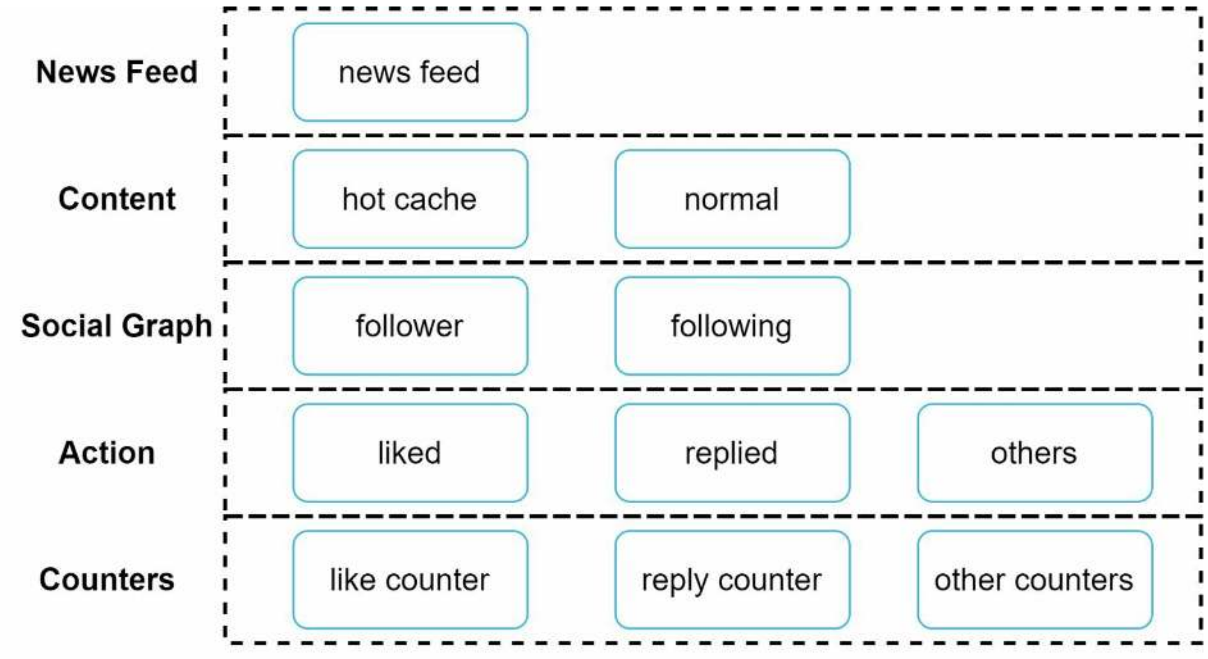

Figure 11-8

• News Feed: It stores IDs of news feeds.
• Content: It stores every post data. Popular content is stored in hot cache.
• Social Graph: It stores user relationship data.
• Action: It stores info about whether a user liked a post, replied a post, or took other actions on a post.
• Counters: It stores counters for like, reply, follower, following, etc.

Step 4 - Wrap up

In this chapter, we designed a news feed system. Our design contains two flows: feed publishing and news feed retrieval.

Like any system design interview questions, there is no perfect way to design a system. Every company has its unique constraints, and you must design a system to fit those constraints. Understanding the tradeoffs of your design and technology choices are important. If there are a few minutes left, you can talk about scalability issues. To avoid duplicated discussion, only high-level talking points are listed below.

Scaling the database:

- Vertical scaling vs Horizontal scaling
- SQL vs NoSQL
- Master-slave replication
- Read replicas
- Consistency models
- Database sharding

Other talking points:

- Keep web tier stateless
- Cache data as much as you can
- Support multiple data centers
- Lose couple components with message queues
- Monitor key metrics. For instance, QPS during peak hours and latency while users refreshing their news feed are interesting to monitor.

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

Reference materials

[1] How News Feed Works:
https://www.facebook.com/help/327131014036297/

[2] Friend of Friend recommendations Neo4j and SQL Sever:
http://geekswithblogs.net/brendonpage/archive/2015/10/26/friend-of-friend-recommendations-with-neo4j.aspx

CHAPTER 12: DESIGN A CHAT SYSTEM

In this chapter we explore the design of a chat system. Almost everyone uses a chat app. Figure 12-1 shows some of the most popular apps in the marketplace.

Figure 12-1

A chat app performs different functions for different people. It is extremely important to nail down the exact requirements. For example, you do not want to design a system that focuses on group chat when the interviewer has one-on-one chat in mind. It is important to explore the feature requirements.

Step 1 - Understand the problem and establish design scope

It is vital to agree on the type of chat app to design. In the marketplace, there are one-on-one chat apps like Facebook Messenger, WeChat, and WhatsApp, office chat apps that focus on group chat like Slack, or game chat apps, like Discord, that focus on large group interaction and low voice chat latency.

The first set of clarification questions should nail down what the interviewer has in mind exactly when she asks you to design a chat system. At the very least, figure out if you should focus on a one-on-one chat or group chat app. Some questions you might ask are as follows:

Candidate: What kind of chat app shall we design? 1 on 1 or group based?
Interviewer: It should support both 1 on 1 and group chat.

Candidate: Is this a mobile app? Or a web app? Or both?
Interviewer: Both.

Candidate: What is the scale of this app? A startup app or massive scale?
Interviewer: It should support 50 million daily active users (DAU).

Candidate: For group chat, what is the group member limit?
Interviewer: A maximum of 100 people

Candidate: What features are important for the chat app? Can it support attachment?
Interviewer: 1 on 1 chat, group chat, online indicator. The system only supports text messages.

Candidate: Is there a message size limit?
Interviewer: Yes, text length should be less than 100,000 characters long.

Candidate: Is end-to-end encryption required?
Interviewer: Not required for now but we will discuss that if time allows.

Candidate: How long shall we store the chat history?
Interviewer: Forever.

In the chapter, we focus on designing a chat app like Facebook messenger, with an emphasis on the following features:

- A one-on-one chat with low delivery latency
- Small group chat (max of 100 people)
- Online presence
- Multiple device support. The same account can be logged in to multiple accounts at the same time.
- Push notifications

It is also important to agree on the design scale. We will design a system that supports 50 million DAU.

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

To develop a high-quality design, we should have a basic knowledge of how clients and servers communicate. In a chat system, clients can be either mobile applications or web applications. Clients do not communicate directly with each other. Instead, each client connects to a chat service, which supports all the features mentioned above. Let us focus on fundamental operations. The chat service must support the following functions:

- Receive messages from other clients.
- Find the right recipients for each message and relay the message to the recipients.
- If a recipient is not online, hold the messages for that recipient on the server until she is online.

Figure 12-2 shows the relationships between clients (sender and receiver) and the chat service.

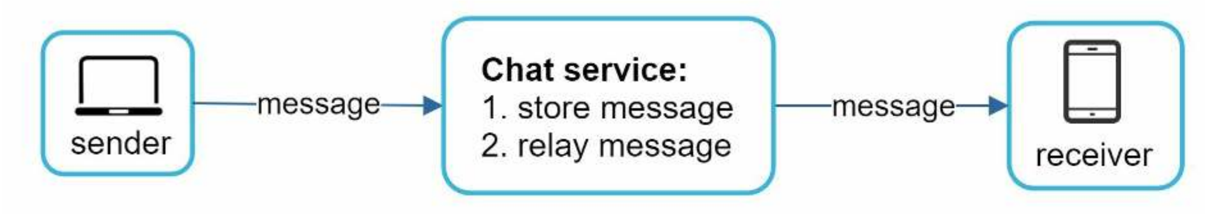

Figure 12-2

When a client intends to start a chat, it connects the chats service using one or more network protocols. For a chat service, the choice of network protocols is important. Let us discuss this with the interviewer.

Requests are initiated by the client for most client/server applications. This is also true for the sender side of a chat application. In Figure 12-2, when the sender sends a message to the receiver via the chat service, it uses the time-tested HTTP protocol, which is the most common web protocol. In this scenario, the client opens a HTTP connection with the chat service and sends the message, informing the service to send the message to the receiver. The keep-alive is efficient for this because the keep-alive header allows a client to maintain a persistent connection with the chat service. It also reduces the number of TCP handshakes. HTTP is a fine option on the sender side, and many popular chat applications such as Facebook [1] used HTTP initially to send messages.

However, the receiver side is a bit more complicated. Since HTTP is client-initiated, it is not trivial to send messages from the server. Over the years, many techniques are used to simulate a server-initiated connection: polling, long polling, and WebSocket. Those are important techniques widely used in system design interviews so let us examine each of them.

Polling

As shown in Figure 12-3, polling is a technique that the client periodically asks the server if there are messages available. Depending on polling frequency, polling could be costly. It could consume precious server resources to answer a question that offers no as an answer most of the time.