12장 - 채팅 시스템 설계

1단계 : 문제 이해 및 설계 범위 확정

- 시장에 나와 있는 앱들을 살펴보면 `페이스북 메신저`, `위챗`, `왓츠앱` 처럼 **1:1 채팅에 집중하는 채팅**들이 있고,

  `슬랙` 같은 **그룹 채팅에 중점을 둔 업무용 앱**이나, `게임 채팅`에 쓰이는 **디스코드** 등 다양하다.
  따라서 초반에 던져야 하는 질문들은
  면접관이 원하는 앱이 정확히 무엇인지 알아내야 한다.

지원자: 어떤 앱을 설계해야 하나요? 1:1 채팅 앱인가요? 아니면 그룹 채팅 앱인가요?

면접관: 둘 다 지원할 수 있어야 합니다.

지원자: 모바일 앱인가요 아니면 웹 앱인가요?

면접관: 둘 다 입니다.

지원자: 처리해야 하는 트래픽 규모는 어느 정도입니까?

면접관: DAU 기준 5천만명을 처리할 수 있어야 합니다.

지원자: 그룹 채팅의 경우에 인원 제한이 있습니까?

면접관: 최대 100명까지 참가할 수 있습니다.

지원자: 중요 기능으로는 어떤 것이 있을까요? 가령, 첨부파일도 지원할 수 있어야 하나요?

면접관: 1:1 채팅, 그룹 채팅, 사용자 접속상태 표시를 지원해야 합니다.
텍스트 메시지만 주고받을 수 있습니다.

지원자: 메시지 길이에 제한이 있나요?

면접관: 네. 100,000자 이하여야 합니다.

지원자: 종단 간 암호화를 지원해야 하나요?

면접관: 현재로서는 필요 없습니다만 시간이 허락하면 논의해볼 수 있겠습니다.

지원자: 채팅 이력은 얼마나 오래 보관해야 할까요?

면접관: 영원히요.

> 페이스북 메신저와 유사한 채팅 앱을 설계하고, 다음과 같은 기능을 갖는다.

- 응답지연이 낮은 일대일 채팅 기능
- 최대 100명까지 참여할 수 있는 그룹 채팅 기능
- 사용자의 접속상태 표시 기능
- 다양한 단말 지원. 하나의 계정으로 여러 단말에 동시 접속 지원
- 푸시 알림
- 5천만 DAU를 처리할 수 있도록 하는 것

2단계 : 개략적 설계안 제시 및 동의 구하기

- 클라이언트와 서버의 통신 방법에 대한 기본적 지식이 있어야 답을 좋은 답을 찾을 수 있다.
- 채팅 시스템의 경우 클라이언트는 모발이 앱이거나 웹 애플리케이션이다.
- 클라이언트는 서로 직접 통신하지 않고,
  각 클라이언트는 위에 나열한 모든 기능을 지원하는 채팅 서비스와 통신한다.

> **채팅 서비스는 아래 기능을 제공해야 함**

- 클라이언트들로부터 메시지 수신

- 메시지 수신자(recipient) 결정 및 전달
- 수신자가 접속(online) 상태가 아닌 경우에는 접속할 때까지 해당 메시지 보관

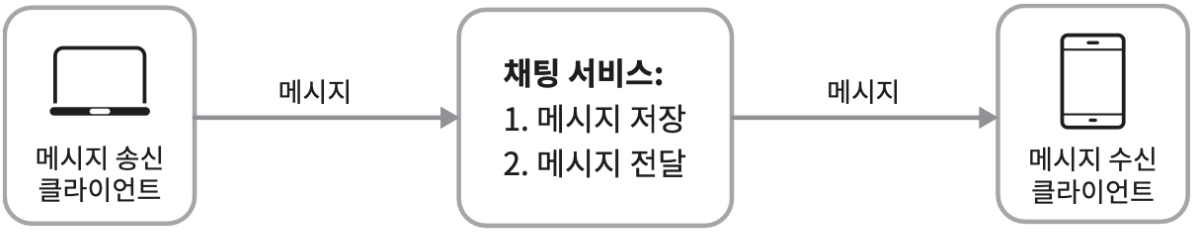

그림 12-2

`송신`과 `수신 클라이언트`와 `채팅 서비스` 사이의 관계를 요약한 그림

- 클라이언트는 네트워크 통신 프로토콜을 사용해 서비스에 접속한다.
  이때
  어떤 통신 프로토콜을 사용할 것인가도 중요한 문제이다. (면접관과 상의하자)
- 위 그림에선 `송신`이 `수신`에게 전달할 메시지를 채팅 서비스에 보낼 때, `HTTP 프로토콜`을 사용한다.
  이때 채팅 서비스와의 접속에는
  `keep-alive` **헤더를 사용**하면 좀 더 효율적이다.
  해당 헤더를 사용하면
  서버 사이의 연결을 끊지 않고 계속 유지할 수 있게된다.
  - 이를 통해 TCP 접속 과정에 발생하는 **핸드셰이크 횟수를 줄일 수** 있음
  - 페이스북 같은 많은 대중적 채팅 프로그램이 **초기에는 HTTP를 사용** ([1] 참고)
- 하지만 메시지 `수신` 시나리오는 이것보다 더 복잡하다.
  - HTTP는 클라이언트가 연결을 만드는 프로토콜, 서버에서 클라이언트로 임의 시점에 메시지를 보내는 데는 쉽게 쓰일 수 없다.
  - 서버가 연결을 만드는 것처럼 동작할 수 있도록 하기 위해 많은 기법이 제안되어 왔음
    - `폴링(polling)`, `롱 폴링(long polling)`, `웹소켓(websocket)` 등

폴링

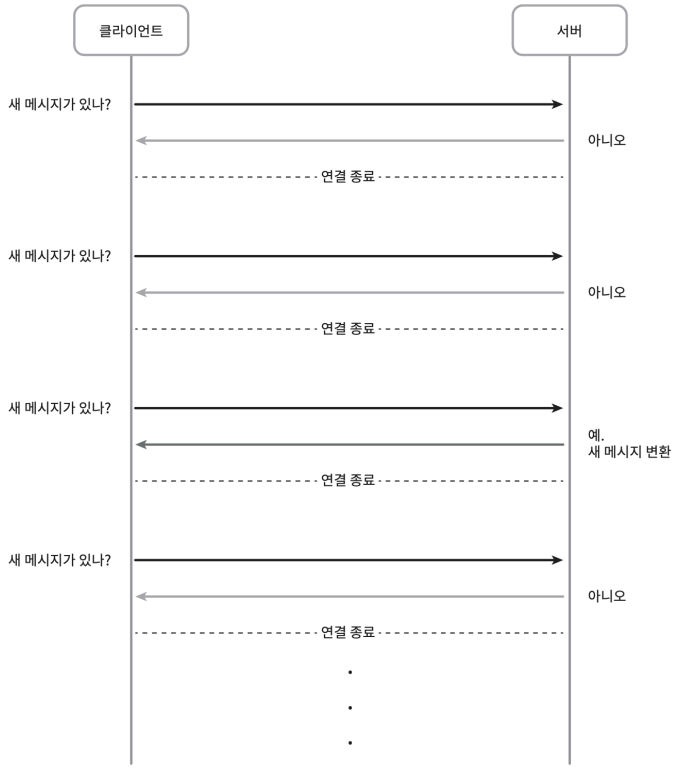

그림 12-3

- 클라이언트가 주기적으로 서버에게 새 메시지가 있는지 물어보는 방법
- 폴링 비용은 폴리을 자주하면 할수록 올라간다.
- **(단점)** - 단, 답해줄 메시지가 없는 경우에는 서버 자원이 불필요하게 낭비됨

롱 폴링

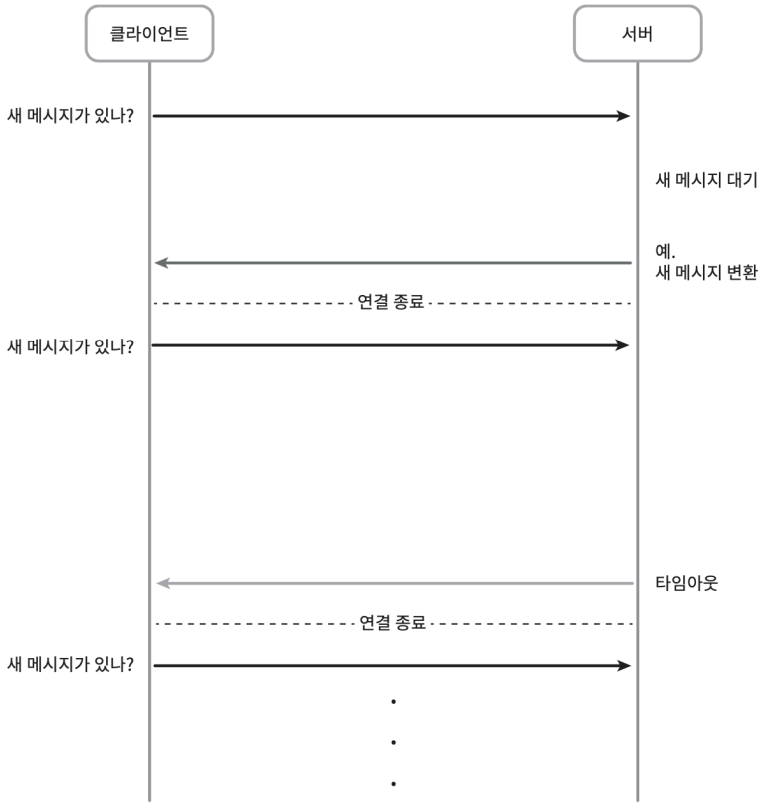

그림 12-4

- 폴링의 비효율을 해결하기 위해 나온 기법
- 클라이언트는 새 메시지가 반환되거나 타임아웃 될 때까지 **연결을 유지**
  - 새 메시지를 받으면 기존 연결을 종료하고, 서버에 새로운 요청을 보내 모든 절차를 다시 시작 함
- **(단점)**
  - 메시지를 `보내는 클라이언트`와 `수신하는 클라이언트`가 **같은 채팅 서버에 접속하게 되지 않을 수** 있음

    HTTP 서버들은 보통 무상태 서버이므로,
    로드밸런싱을 위해 라운드 로빈 알고리즘을 사용하는 경우
    메시지를
    `받은 서버`는 `수신할 클라이언트`와의 롱 폴링 연결을 가지고 있지 않는 서버일 수 있음
  - 서버 입장에서는 클라이언트가 연결을 해제했는지 아닌지 알 좋은 방법이 없음
  - 여전히 비효율적임

    메시지를 많이 받지 않는 클라이언트도 타임아웃이 일어날 때마다 주기적으로 서버에 다시 접속

웹소켓

- 웹 소켓은 서버가 클라이언트에게 **비동기(async) 메시지를 보낼 때 가장 널리 사용**하는 기술이다.

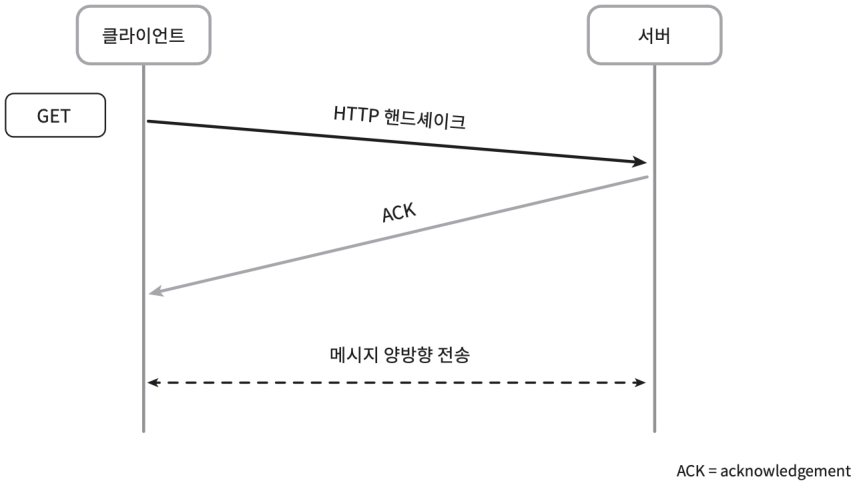

그림 12-5

- 웹소켓 연결은 클라이언트가 시작한다.
- 한 번 맺어진 연결은 항구적(= 영구적)이며 양방향이다.
- 이 연결은 **처음에는 HTTP 연결**이지만 **특정 핸드셰이크 절차를 거쳐 웹소켓 연결로 업그레이드**된다.
- 이렇게 영구적인 연결이 만들어지고 나면 서버는 클라이언트에게 비동기적으로 메시지를 전송할 수 있다.
- 일반적으로 방화벽이 있는 환경에서도 잘 동작한다.
  - 80, 443, HTTP 또는 HTTPS 프로토콜이 사용하는 **기본 포트번호를 그대로 사용**함
- 요약하면 **웹소켓은 HTTP 프로토콜에서 양방향 메시지 전송까지 가능**하게 된다.
  - 즉, 웹소켓 대신 HTTP를 고집할 이유는 없음, 웹 소켓 쓰자

> 어떻게 웹소켓이 메시지 전송이나 수신에 쓰일 수 있는지 간단히 살펴보기

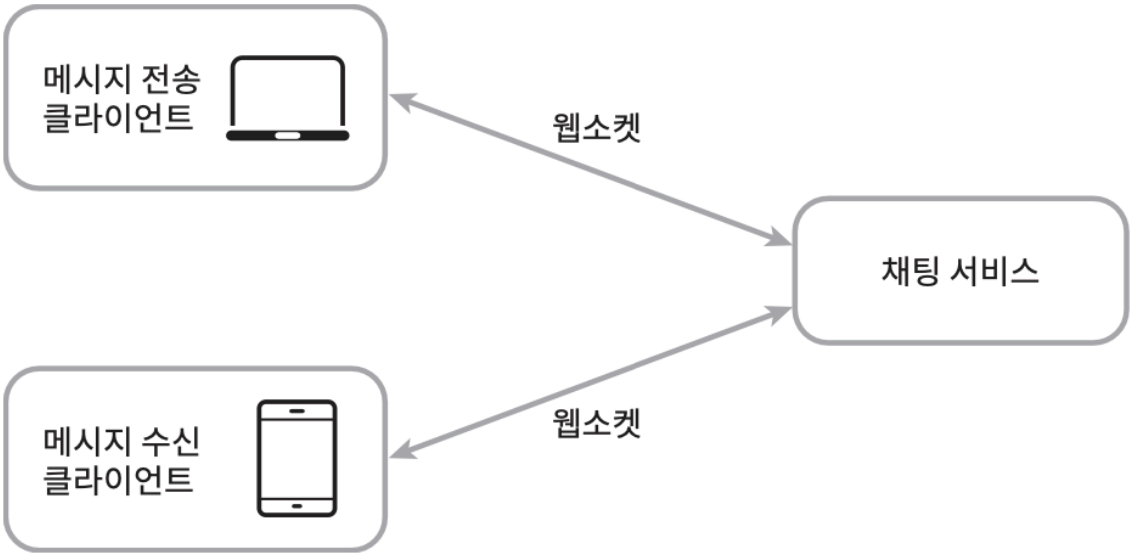

그림 12-6

- 웹 소켓을 이용하면 메시지를 보낼 때나 받을 때 **동일한 프로토콜**을 사용할 수 있음
  - 설계뿐 아니라 구현도 단순하고 직관적임
  - **(조심)** 웹소켓은 연결이 항구적으로 유지되기 때문에 **서버 측에서 연결 관리를 효율적으로 해야 됨**

> **다른 부분에서도 웹소켓을 사용하는 것이 좋을까?**

- 굳이 웹소켓을 사용할 필요가 없음
- 대부분은 기능(회원가입, 로그인, 사용자 프로필 등)은 일반적인 HTTP 상에서 구현해도 됨
- 이 부분에 대해서 좀 더 찾아볼까?

개략적 설계안

- 이번 장에서 다루는 채팅 시스템은 세 부분으로 나눌 수 있다.
  - `무상태 서비스`, `상태유지(stateful) 서비스`, `제3자 서비스 연동`

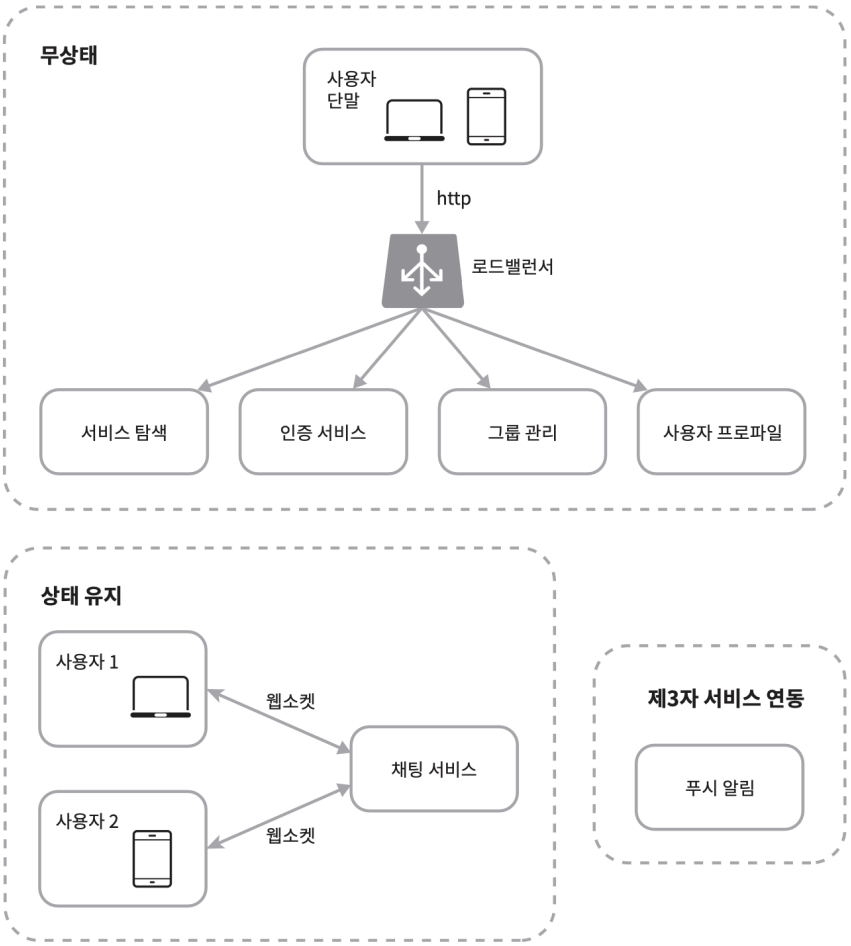

그림 12-7

무상태(sateless) 서비스

- 로그인, 회원가입, 사용자 프로파일 표시 등을 처리하는 전통적인 요청/응답 서비스
  - 제공하는 기능은 많은 웹사이트와 앱이 **보편적으로 제공하는 기능**.
- 로드밸러서 뒤에 위치한다.
  - 로드밸런서가 하는 일은 요청을 그 경로에 맞는 서비스로 정확하게 전달하는 것

    로드밸런서 뒤에 오는 서비스는 모놀리틱(monolithic) 서비스 일 수 있고, MSA일 수 있음
- 이 서비스는 클라이언트가 접속할 채팅 서버의 DNS 호스트명을 클라이언트에게 알려주는 역할을 함

상태유지(stateful) 서비스

- 위 그림에서 본 설계안에서 **유일하게 상태 유지가 필요한 서비스는 채팅 서비스**이다.
  - **각 클라이언트**가 **채팅 서버와 독립적인 네트워크 연결을 유지**해야 하기 때문
- 클라이언트는 보통 서버가 살아 있는 한 다른 서버로 연결을 변경하지 않는다.
  - `서비스 탐색 서비스`는 **채팅 서비스**와 **협력**하여 **특정 서버에 부하가 몰리지 않도록** 해야 됨

제3자 서비스 연동

- 채팅 앱에서 가장 중요한 제3자 서비스는 **푸시 알림**이다.
  - 새 메시지를 받았다면 앱이 실행 중이지 않더라도 알림을 받아야 함

규모 확장성

- 트래픽 규모가 작으면 서버 한 대로 모두 구현 가능
- 대량의 트래픽을 처리해야 하는 경우라면?
  - 동시 접속자가 1M이라고 가정하고, 접소당 10K의 서버 메모리가 필요하다면 10KB 메모리만 있으면 모든 연결을 다 처리할 수 있음
  - 이때 서버 한 대로만 처리할 수 있지만, SPOF 같은 상황으로 인해, 좋은 점수를 받지는 못 함

여기까지 설계한 그림

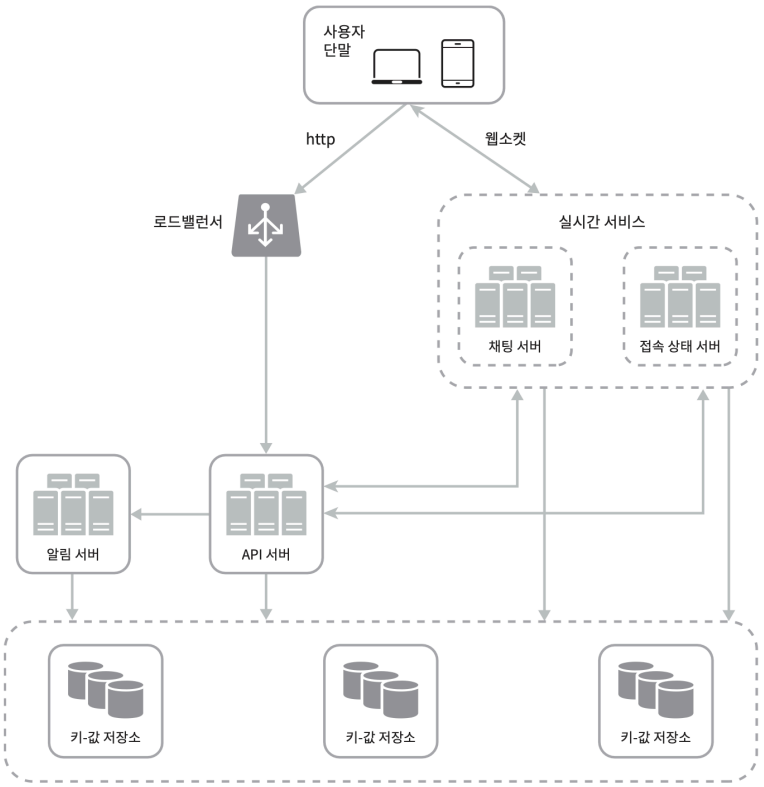

그림 12-8

- 실시간으로 메시지를 주고받기 위해 클라이언트는 채팅 서버와 웹소켓 연결을 끊지 않고 유지해야 됨
- 채팅 서버는 클라이언트 사이에 메시지를 중계하는 역할 담당
- 접속상태 서버(presence server)는 사용자의 접속 여부를 관리
- API 서버는 로그인, 회원가입, 프로파일 변경 등 그 외 나머지 전부 처리
- 알림 서버는 푸시 알림을 보냄
- 키-값 저장소에는 채팅 이력(chat history)을 보관
  - 시스템에 접속한 사용자는 이전 채팅 이력을 전부 보게 됨

저장소 - 어떤 데이터베이스를 쓸까? NoSQL vs RDB?

- 어떤 저장소를 고를 때 중요한 것은 **데이터의 유형과 읽기/쓰기 연산의 패턴**이 중요
- 채팅 시스템이 다루는 데이터 두 가지
  1. 사용자 프로파일, 설정, 친구 목록처럼 **일반적인 데이터** - **관계형 데이터베이스**
     - 이런 데이터는 **안정성을 보장**하는 **관계형 데이터베이스에 보관**

- **다중화(replication)**와 **샤딩(sharding)**으로 **가용성**과 **규모확장성을 보증**할 수 있음

2. 채팅 시스템에 고유한 데이터인 **채팅 이력(chat history) 데이터**이다. - **키-값 저장소**
   - 채팅 이력 데이터에 대해서는 **읽기/쓰기 연산 패턴을 이해**해야 함
     - 채팅 이력 데이터의 양은 많음 ([2] 참고, 페이스북과 왓츠앱은 매일 600억 개의 메시지 처리)
       - 이 중 빈번하게 사용되는 데이터는 **최근에 주고받은 메시지**. (오래된 메시지는 잘 안 봄)
     - 사용자는 대체로 `최근 메시지`, `검색`, `특정 사용자의 언급(mention)`, `특정 메시지로 점프(jump)` 등 **무작위적인 데이터를 접근을 하게 되는 일**도 있음. → 이런 기능도 지원해야 함
     - 1:1 채팅 앱의 경우 읽기:쓰기 비율은 대략 1:1 정도
     - 이 모두를 지원할 데이터베이스로 **키-값 저장소 선택**
   - **키-값 저장소 선택 이유**
     - 키-값 저장소는 **수평적 규모확장(horizontal scaling)**이 **쉬움**
     - 키-값 저장소는 **데이터 접근 지연시간(latency)이 낮음**
     - **RDB는 데이터 중 롱테일(long tail)**에 해당하는 부분을 **잘 처리하지 못하는** 경향이 있음 - [3]
       - 인덱스가 커지면 데이터에 대한 무작위적 접근(random access)을 처리하는 비용이 늠
     - 이미 많은 안정적인 채팅 시스템이 키-값 저장소를 채택하고 있음
       - ex) 페이스북 메신저, 디스코드. 페이스북 HBase 사용, 디스코드 카산드라 사용

데이터 모델 - `키-값 저장소`를 사용하기로 했으며, 메시지 데이터를 어떻게 보관할까?

1:1 채팅을 위한 메시지 테이블

message	
message_id	**bigint**
message_from	bigint
message_to	bitint
content	text
created_at	timestamp

그림 12-9

1:1 채팅을 위한 메시지 테이블

기본 키는 `message_id`로 메시지 순서를 쉽게 정할 수 있는 역할도 담당.
서로 다른 두 메시지가 동시에 만들어질 수 있기 때문에
`created_at`으로 메시지 순서를 정할 수는 없음

그룹 채팅을 위한 메시지 테이블

group message	
channel_id	**bigint**
message_id	**bigint**
message_to	bitint
content	text
created_at	timestamp

그림 12-10

그룹 채팅을 위한 메시지 테이블

- (`channel_id, message_id`)의 복합 키를 PK로 사용
  - 채널은 채팅 그룹과 같은 뜻

- `channel_id`는 파티션 키로도 사용할 것이고, **그룹 채팅에 적용될 모든 질의는 특정 채널을 대상**으로 함

메시지 ID

- messaged_id는 메세지들의 순서도 표현할 수 있어야 됨으로 다음의 속성을 만족해야 함
  - messaged_id의 값은 고유해야 함 (uniqueness)
  - ID 값은 정렬 가능해야 하며 시간 순서와 일치해야 함 (새로운 ID는 이전 ID보다 큰 값이어야 함)
- 위 속성을 만족시키기 위해선 `RDB` **라면** `auto_increment`가 대안이지만, **NoSQL은 해당 기능이 없다.**
  - 스노플레이크 같은 전역적 64-bit 순서 번호 생성기를 이용할 수 있음.
  - 다른 방식으로는 **지역적 순서 번호 생성기(local sequence number generator)**를 이용하는 방법
    - 지역적이라 함은 ID와 유일성은 같은 그룹 안에서만 보증하면 충분하다는 내용

3단계 : 상세 설계

- 개략적 설계안에 포함된 컴포넌트 중 몇 가지를 골라 자세히 들여다보자.
- 채팅 시스템의 경우 `서비스 탐색`, `메시지 전달 흐름`, `사용자 접속 상태`를 표시하는 방법 정도가 있음

서비스 탐색 (service discovery)

- 이 기능의 주된 역할은 클라이언트에게 가장 적합한 채팅 서버를 추천한다.
  - **추천 기준**으로는 **클라이언트의 위치(geographical location)**, **서버의 용량(capacity)** 등이 있다.
- 이 기능을 구현하는 데 널리 쓰이는 오픈 소스 솔루션으로는 아파치 주키퍼 같은 것이 있다.

- **사용 가능한 모든** 채팅 서버를 여기 **등록**시키고,

  클라이언트가 접속을 시도하면 사전에 정한 기준에 따라 **최적의 채팅 서버를 골라 주면** 됨

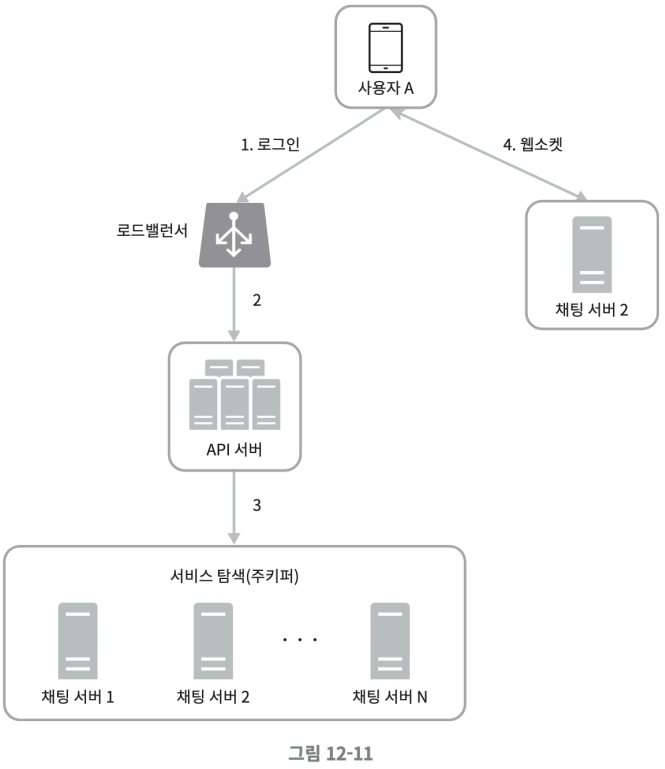

그림 12-11

주키퍼로 구현한 서비스 탐색 기능

1. 사용자 A가 시스템에 로그인 시도
2. 로드밸런서가 로그인 요청을 API 서버들 가운데 하나로 보냄
3. API 서버가 사용자 인증을 처리하고 나면 서비스 탐색 기능이 동작하여 해당 사용자를 서비스할 최적의 채팅 서버를 찾음. (위 예제는 `채팅 서버2`가 선택되어 반환 됨)
4. 사용자 A는 채팅 서버 2와 웹소켓 연결을 맺음

메시지 (전달) 흐름

- 종단 간 메시지 흐름을 이해하기 위해 다음의 주제들을 살펴봄
  - 1:1 채팅 메시지 처리 흐름
  - 여러 단말 사이의 메시지 동기화

- 소규모 그룹 채팅에서의 메시지 흐름

1:1 채팅 메시지 처리 흐름

- 다음 그림은 1:1 채팅에서 사용자 A가 B에게 보낸 메시지가 어떤 경로로 처리되는지 보여준다.

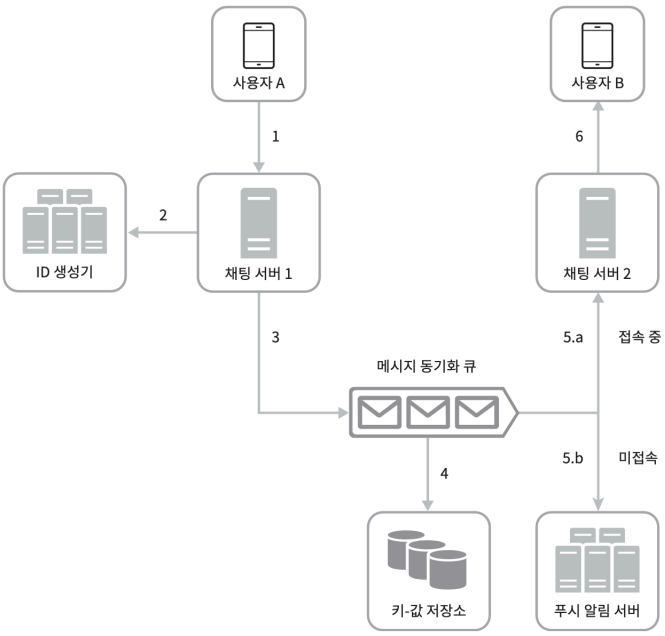

그림 12-12

1. 사용자 A가 채팅 서버 1로 메시지 전송
2. 채팅 서버 1은 ID 생성기를 사용해 해당 메시지의 ID 결정
3. 채팅 서버 1은 해당 메시지를 메시지 동기화 큐로 전송
4. 메시지가 키-값 저장소에 보관됨
5. 채팅을 받는 **사용자 B가 채팅 서버에 접속 여부**에 따라 달라짐
    a. **접속 중**인 경우 메시지는 사용자 B가 접속 중인 **채팅 서버**(이 예제의 경우 채팅 서버 2)로 **전송**
    b. **접속 중이 아니라면 푸시 알림** 메시지를 푸시 알림 서버로 보냄
6. 채팅 서버 2는 메시지를 사용자 B에게 전송.
    - 사용자 B와 채팅 서버 2사이에는 웹소켓 연결이 있는 상태로 웹소켓을 이용

여러 단말 사이의 메시지 동기화 - 1:1 채팅에 비해 더 복잡

소규모 그룹 채팅에서 사용

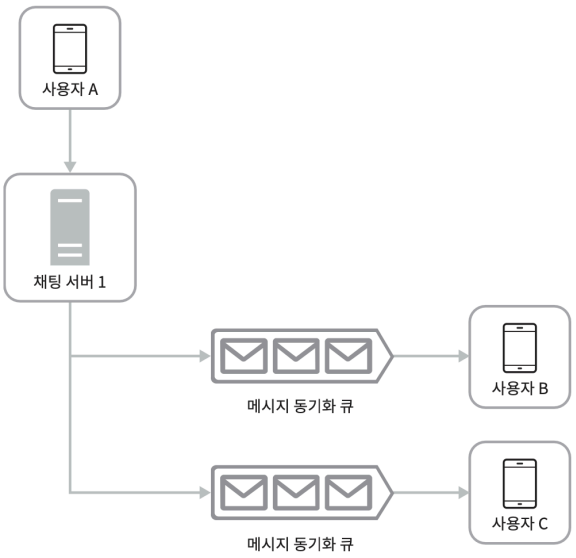

그림 12-14

- 위 그림은 사용자 A가 그룹 채팅 방에서 메시지를 보냈을 때 어떤 일이 벌어지는지 보여준다.
  - 해당 그룹에 3명의 사용자가 있다고 가정(사용자 A, B, C)
  - 사용자 A가 보낸 메시지가 사용자 B와 C의 메시지 동기화 큐(message sync queue)에 복사
    - 사용자 각각에 할당된 메시지 수신함 같은 것으로 생각해도 됨
  - 이런 방식은 `소규모 그룹 채팅`에 적합
    - 새로운 메시지가 왔는지 확인하려면 자기 큐만 보면 되니까 메시지 동기화 플로가 단순
    - 그룹이 크지 않으면 메시지를 수신자별로 복사해서 큐에 넣는 작업의 비용이 문제가 되지 않음
    - `위챗`이 이런 접근법을 사용하고 있고, **그룹의 크기는 500명으로 제한**하고 있음 - [8] 참고
  - 소규모가 적합한 이유는 **똑같은 메시지를 모든 사용자의 큐에 복사하는 것이 바람직하지 않다**.

수신자 관점에서 살펴본 그림 흐름

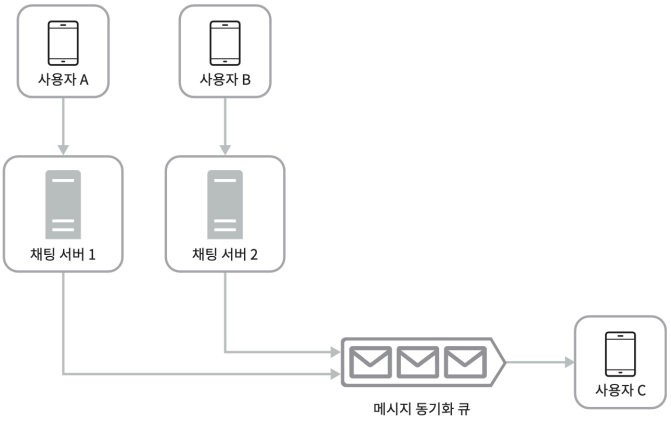

그림 12-15

- 한 수신자는 여러 사용자로부터 오는 메시지를 수신할 수 있어야 한다.
- 각 사용자의 수신함, 즉 메시지 동기화 큐는 위 그림처럼 여러 사용자로부터 오는 메시지를 받을 수 있어야 됨

접속상태 표시 (사용자 접속상태)

- 개략적 설계안에서는 접속상태 서버(presense server)를 통해 사용자의 상태를 관리한다고 했었다.
- 접속상태 서버는 클라이언트와 웹소켓으로 통신하는 실시간 서비스의 일부라는 점이다.

사용자 로그인

그림 12-16

- 클라이언트와 실시간 서비스(real-time service) 사이에 웹소켓 연결이 맺어지고 나면 접속상태 서버는 A의 상태와 last_active_at 타입트새픔 값을 키-값 저장소에 보관한다.
- 이 절차가 끝나고 나면 해당 사용자는 접속 중인 것으로 표시될 수 있다.

로그아웃

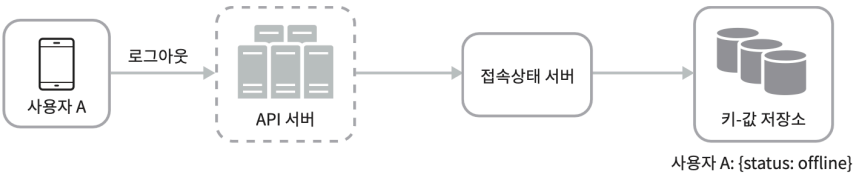

그림 12-17

- 키-값 저장소에 보관된 사용자 상태가 online에서 offline으로 바뀌게 된다.
- 이 절차가 끝나고 나면 UI 상에서 사용자의 상태는 접속 중이 아닌 것으로 표시될 것이다.

접속 장애

- 사용자의 인터넷 연결이 끊어지면 클라이언트와 서버 사이에 맺어진 웹소켓 같은 지속성 연결도 끊어진다.
  - 간단한 대응으로는 사용자를 오프라인 상태로 표시하고, 연결이 복구되면 온라인으로 변경하면 된다.
  - 하지만 이 방법의 문제는 짧은 시간 동안 **인터넷 연결이 끊어졌다 복구**되는 일은 **흔하다**.
    ex) 사용자가 차를 타고 터널을 지나가는 상황같은, 이럴 때마다 접속 상태를 바꾸면 지나친 일이 된다.
- 본 설계안에서는 **박동(heartbeat) 검사**를 통해 이 문제를 해결한다.
  - `온라인 상태의 클라이언트`로 하여금 **주기적으로 박동 이벤트를 접속상태 서버로 보내도록** 하고,

    마지막 이벤트를 받은 지 x초 이내의 또 다른 박동 이벤트 메시지를 받으면
    해당 사용자는 계속 온라인으로 유지하고, 그렇지 않을 경우에만 오프라인으로 변경한다.
- 다음 그림의 예제에서는 박동 이벤트를 매 5초마다 서버로 보내고 있다.
  이때 이벤트를 3번 보낸 후, x = 30초 동안 아무런 메시지를 보내지 않아 오프라인으로 변경되었다.

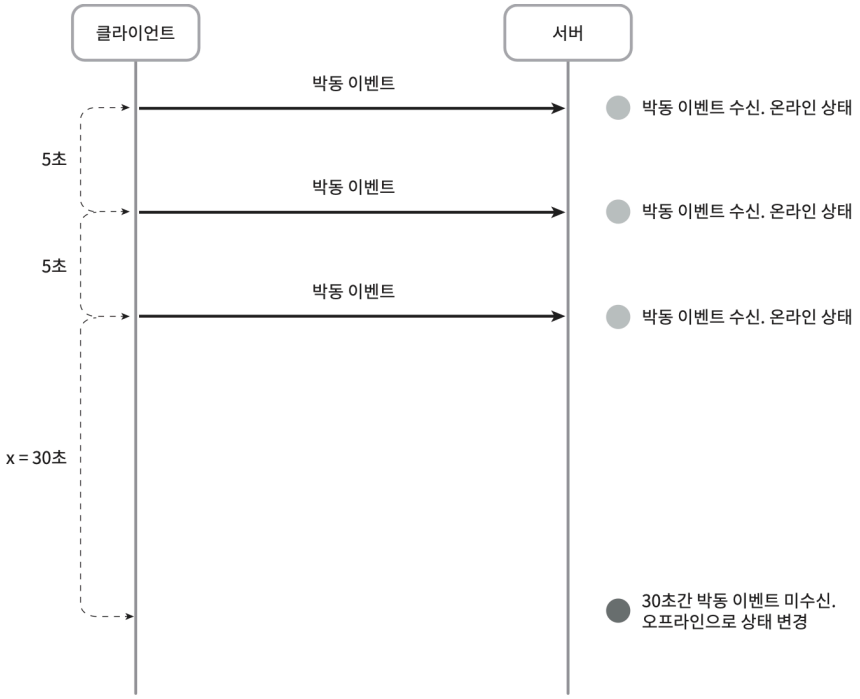

그림 12-18

사용자 A와 친구 관계에 있는 사용자들은 해당 사용의 상태 변화를 알게 될까?

상태 정보의 전송

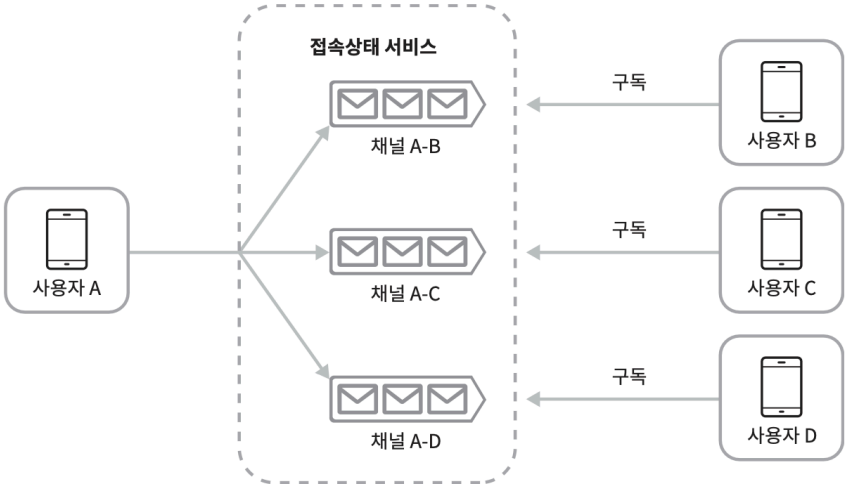

그림 12-19

- 상태정보 서버는 발행-구독 모델(publish-subscribe model)을 사용한다.
  - 즉, 각각의 친구관계마다 채널을 하나씩 두는 것이다.
  - ex) 사용자 A의 접속상태가 변경되었다고 하자.

        - 그 사실을 **세 개 채널**, 즉 `A-B`, `A-C`, `A-D`에 쓰는 것이다.
          그리고
          `A-B`는 **사용자 B**가 구독하고, `A-C`는 **사용자 C**가, 그리고 `A-D`는 **사용자 D**가 구독하도록
        - 이렇게하면 친구 관계에 있는 사용자가 상태정보 변화를 쉽게 통지 받을 수 있다.
        - 클라이언트와 서버 사이의 통신에는 실시간 웹소켓 사용
    - 이 방안은 그룹 크기가 작을 때 효과적이다. (위챗은 그룹 크기 상한을 500으로 제한하고 있어 가능)
    - 그룹 하나에 100,000 사용자가 있다면 상태변화 1건당 100,000개의 이벤트 메시지가 발생한다.
- 성능 문제의 해결법으론 사용자가 **그룹 채팅에 입장하는 순간에만 상태 정보를 읽어가게** 하거나,
  친구 리스트에 있는 사용자의 접속 상태를 갱신하고 싶으면 수동으로(manual) 하도록 유도하면 좋다.

4단계 : 마무리

- 클라이언트와 서버 사이의 실시간 통신이 가능하도록 웹소켓 사용
- 실시간 메시징을 지원하는 `채팅 서버`, `접속 상태 서버`, `푸시 알림 서버`, 채팅 이력을 보관할 `키-값 저장소`,
  이를 제외한 나머지 기능을 구현하는데 쓰일
  `API 서버` 등이 **주요 컴포넌트**이다.

추가로 논의해볼 내용

- 채팅 앱을 확장하여 **사진**이나 **비디오** 등의 **미디어를 지원**하도록 하는 방법
    - 미디어 파일은 텍스트에 비해 크기가 크다.
      이와 관련된 압축 방식, 클라우드 저장소, 섬네일(thumbnail) 생성 등을 논의해보면 좋다.
- **종단 간 암호화**
    - 왓츠앱은 메시지 전송에 있어 종단 간 암호화를 지원하낟.

    - 메시지 발신인과 수신자 이외에는 아무도 메시지 내용을 볼 수 없다 - [9]
- **캐시**
    - 클라이언트에 이미 읽은 메시지를 캐시해 두면 서버ㅗ아 주고받는 데이터 양을 줄일 수 있음
- **로딩 속도 개선**
    - 슬랙은 사용자의 데이터, 채널 등을 지역적으로 분산하는 네트워크를 구축해 앱 로딩 속도를 개선 - [10]
- **오류 처리**
    - **채팅 서버 오류**
        - 채팅 서버 하나에 수십만 사용자가 접속해 있는 상황일 때 해당 서버가 죽으면 서비스 탐색 기능(주키퍼 같은)이 동작하여 클라이언트에게 새로운 서버를 배정하고 다시 접속할 수 있도록 해야 한다.
    - **메시지 재전송**
        - 재시도(retry)나 큐(queue)는 메시지의 안정적 전송을 보장하기 위해 흔히 사용되는 기법

Chapter 12: DESIGN A CHAT SYSTEM

1. Erlang at Facebook
2. Messenger and WhatsApp process 60 billion messages a day
3. Long tail
4. The Underlying Technology of Messages
5. How Discord Stores Billions of Messages
6. Announcing Snowflake
7. Apache Zookeeper
8. From nothing: the evolution of WeChat background system
9. End-to-end encryption
10. Flannel: An Application-Level Edge Cache to Make Slack Scale