대용량 트래픽 처리를 위한 채팅 구조 설계

홍성문*, 이윤재*, 고세영**, 정승우***
*한성대학교 컴퓨터공학부
**성결대학교 컴퓨터공학과
***부경대학교 컴퓨터공학과
e-mail : hsm63746244@gmail.com

Design of Chatting Architecture that Handle Large-Scale Traffic

Seong-Mun Hong*, Yoon-jae Lee*, Se-Young Ko **, Seung-Woo Jung***
*Dept. of Computer Engineering, Han-Sung University
**Dept. of Computer Engineering, Sung-Kyul University
***Dept. of Computer Engineering, Pu-Kyong University

요 약

웹 서비스의 트래픽은 변화의 폭이 크다. 또한 서비스는 실시간으로 변화하는 트래픽에 대비하기 위하여 트래픽의 최대치를 가정하여 서버를 구성해야한다. 하지만 트래픽의 최대치와 평균적인 트래픽은 큰 차이가 있어 위와 같은 서버 구성은 많은 자원의 낭비로 이어진다. 이렇듯 실시간으로 변화하는 트래픽에 대응하기 위하여 분산 시스템 구조와 InMemory Cache, Messaging Queue 등을 활용하여 대응하도록 설계했다. 또한 InMemory Cache 와 NoSQL 을 활용하여 효과적으로 메세지를 저장하고 검색할 수 있도록 설계하였다.

1. 서론

최근 휴대폰 사용량 증가에 따라 웹/앱 서비스의 트래픽이 급상승하고 있다. 이러한 웹/앱 서비스의 트래픽은 그 시점의 이슈, 사건, 사고와 밀접하게 관련된다.

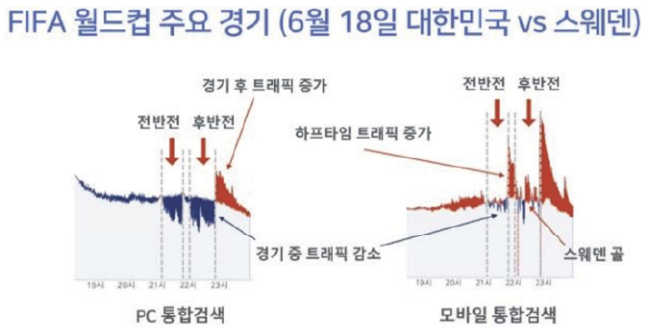

그림 1. 2018 년 월드컵 경기에 따른 트래픽 변화

그림 1 은 네이버에서 발표한 월드컵 한국 경기에 따른 검색 트래픽의 변화를 나타낸다. 또한 지진이 난 후 평소에 3 배 가량 많은 검색 트래픽이 발생했다고 한다. [1] 이렇듯 웹 서비스의 트래픽은 예측할 수 없으며, 변화의 폭이 크다. 이러한 큰 트래픽의 변화에 따라 기존의 모놀리식(Monolithic) 아키텍처를 따른다면 소프트웨어의 구조적인 Coupling 이 강해져 변화에 대응하기 어렵다. 또한 지속적으로 기능이 추가되는 웹 서비스의 특성상 유지보수에 어려움이 따른다. 또한 모놀리식 설계는 Scale Out 하기 어려우며 서버를 분산시키더라도 모든 기능을 포함한 서버가 여러 대 동작하여 효율적이지 못하다.

위와 같은 문제점을 해결하고자 마이크로 서비스 아키텍처의 구조를 갖도록 서버를 분리하여 설계하였으며, InMemory Cache(Redis)와 Messaging Queue(Kafka)을 통해 효율적으로 자원을 사용하도록 설계하였다. 또한 NoSQL 을 활용하여 메세지를 빠르고 효율적으로 저장 및 탐색 할 수 있도록 설계하였다.

본 논문의 구성은 다음과 같다. 2 장에서는 전반적인 프로젝트의 구조를 제시한다. 3 장에서는 본 논문의 핵심인 채팅 구조를 설명하고 4 장에서는 결론을 맺는다.

2. 프로젝트 아키텍처 및 흐름

본 논문에서 설계한 프로젝트의 전반적인 구조는 그림 2 와 같다. 클라이언트는 ReactJS 를 활용하였으며, 서버는 기능에 따라 인증, API, 파일, 채팅, 푸시 서버로 분리하였다. 인증 서버는 로그인 시 Token 을 발급하여 Redis 에 저장하고 세션을 유지시키고 삭제한다. 또한 멤버십 관련 모든 기능을 포함한다. API 서버는 채팅, 파일 전송 이외의 API 들을 포함한다. 파일 서버는 파일을 보내거나 받는 기능을 담당하며 채팅 서버는 STOMP 을 활용하여 채팅 메세지 전송 역할을 담당한다. 푸시 서버는 수신되는 메세지를 전달하는 역할을 담당한다.

사용자가 로그인을 하면 STOMP(Streaming Text Oriented Messaging Protocol)[2]와 SSE(Server Sent Event)[3]에

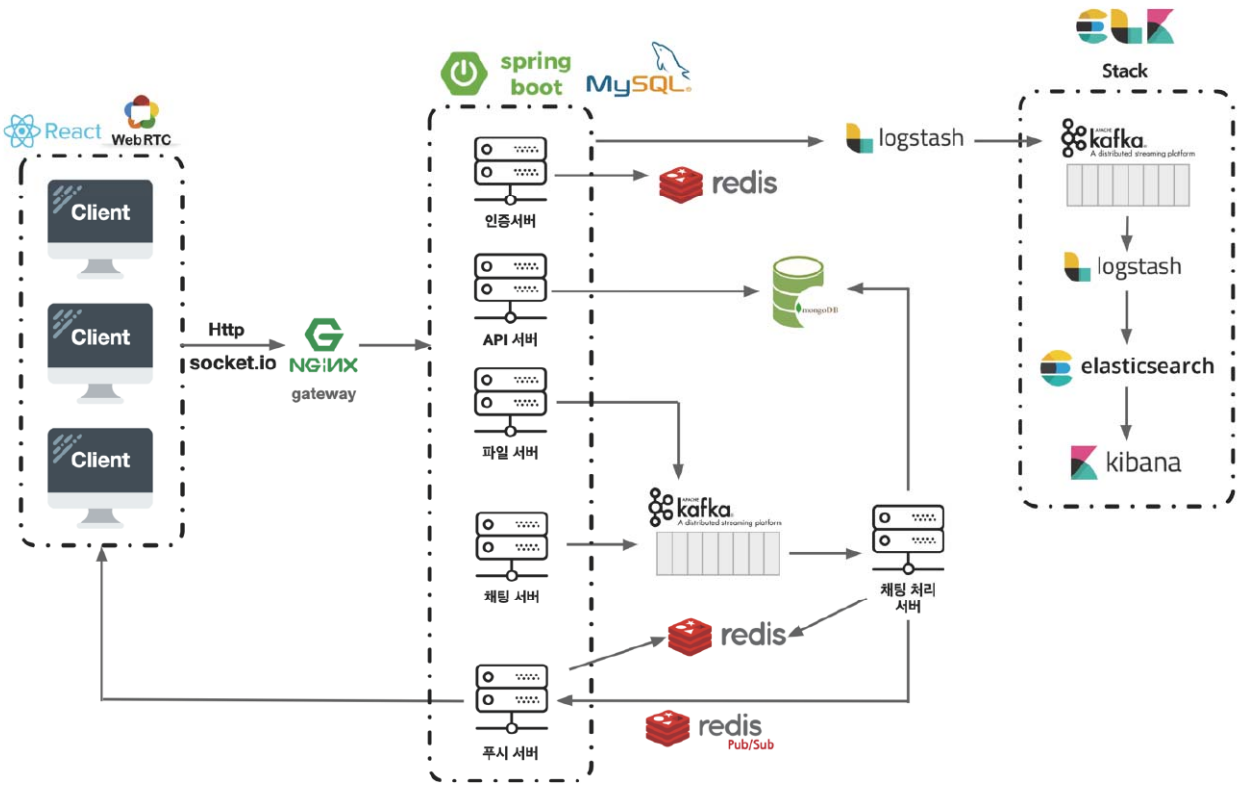

그림 2. 프로젝트의 전반적인 구조

연결을 하게 되고, 사용자가 특정 채팅방을 선택해 채팅방에 접속하게 되면 해당 채팅방의 채널을 구독하게 되며 실시간으로 채팅방 메세지를 수신한다. 또한 채팅 메세지를 보내면 채팅 서버는 서비스 로직 없이 Kafka Topic 에 메세지를 전달하게 되며 채팅 처리 서버는 Topic 의 데이터를 받아 메세지를 처리한 후 푸시 메세지 전송을 위하여 Redis 로 Publish 한다. Subscribe 받은 푸시 서버는 해당 메세지를 사용자에게 전달한다.

3. 대용량 트래픽 처리를 위한 채팅 구조 설계

3-1. 메세지 수신과 송신(STOMP, SSE)

본 논몬의 채팅 메세지의 송수신에는 STOMP 와 SSE 를 사용하였다. 채팅을 전송하는 채팅 서버와 메세지를 수신하는 푸시 서버를 나누어 설계하였다. 다른 API 에 비해 상대적으로 많이 호출될 API 이기 때문에 서버를 분리하여 설계하였다. 또한 한 채팅방의 메세지를 수신하는 것이 아니라 실시간으로 포함되어 있지 않던 새로운 채팅방에 초대되는 경우와 같은 상황을 고려할 때 STOMP 의 Subscribe 기능을 사용한다면 모든 사용자가 초대 정보와 관련된 Topic 을 구독하고 있어야 한다.

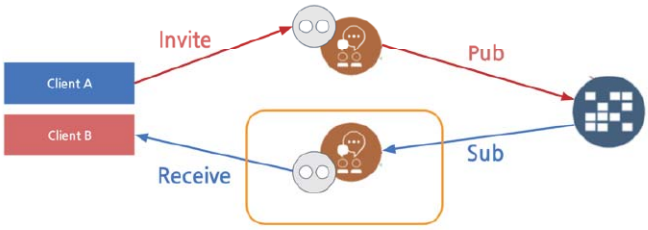

그림 3. STOMP 를 활용한 초대 메세지 수신

그림 3 의 주황색 상자 영역과 같이 사용자 A, B 만 존재하는 것이 아니라 접속한 사용자 모두가 정보를 주고받을 한 Topic 을 구독하는 형태가 되어 결국 메세지 전송에 모든 사용자를 순회해야 하게 된다.

이러한 문제점을 해결하기 위하여 SSE 를 선택하였다. 최초 로그인시 SSE 연결을 진행한 후 초대 메세지를 받게 된다면 초대 메세지를 받는 사용자에게 SSE 를 통해 메세지를 바로 전달하게 된다. 이러한 경우 특정 토픽을 모두 구독할 필요가 없으며, 사용자의 ID 와 같은 값을 통해 사용자를 식별한 후 초대 메세지를 바로 전달할 수 있다.

3-2. Kafka / Redis Pub&Sub 를 이용한 서버간 통신

채팅은 다른 어떠한 API 보다 트래픽이 많은 서버가 될 것이라고 생각된다. 가장 많은 데이터를 저장하고 처리하게 될 것이다. 이러한 경우 더욱 효율적으로 메세지를 처리하기 위하여 Kafka 와 Redis 의 Pub/Sub 기능을 활용하였다. 우선 채팅 서버는 송신된 메세지를 Kafka 의 Topic 에 쌓는 역할만 한다. 이후

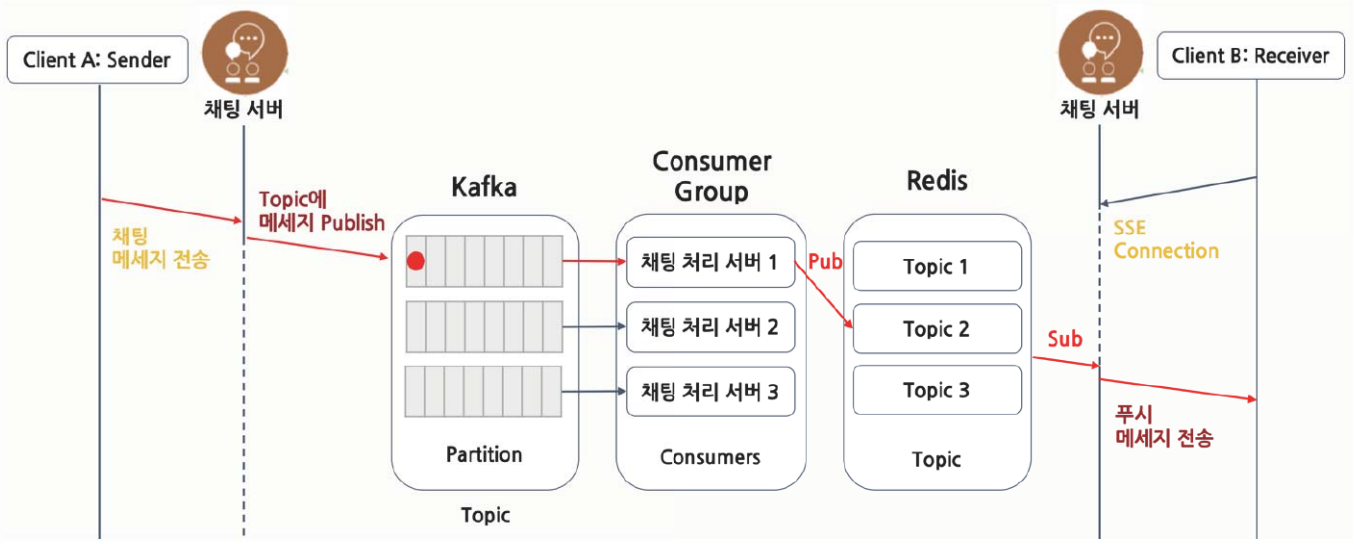

그림 4. 메세지 전송, 수신 세부 흐름

Topic 의 Partition 에 저장된 데이터는 Consumer 역할을 하는 채팅 처리 서버에서 데이터를 받게 되고 받은 데이터의 메세지 처리를 위한 비지니스 로직을 실행한다. 메세지 처리가 끝난 후 푸시 알림 전송을 위해 Redis 에 Pub/Sub 기능을 활용하여 해당 채팅방 Topic 에 메세지를 Publish 한다. 푸시 서버는 해당 채팅방에 존재하는 사용자에게 메세지를 전송한다. A 사용자가 B 사용자에게 메세지를 전달하는 과정은 그림 4 와 같다.

데이터의 처리에 시간이 오래 걸려 채팅 처리 서버에 부하가 생긴다면 그림 4 와 같이 Partition 을 확장하여 채팅 처리 서버를 증설하여 더 효과적으로 메세지를 처리할 수 있게 된다.

3-3. NoSQL 과 Redis 를 활용한 메세지 저장 구조

RDB 를 사용하여 메세지를 저장하게 된다면 하나의 메세지 전송 요청 당 하나의 쿼리를 통하여 메세지를 저장할 수 있다. Redis 와 같은 Memory Cache 를 활용하여 덩어리로 저장한다고 가정하여도 채팅방 별 테이블이 존재하지 않는 이상 식별자 비교만을 통하여 덩어리를 가져올 수 없으며 채팅방 식별자를 확인해야 하며 채팅방 별 테이블을 생성하는 구조는 더 비효율적이다.

메신저 특성상 과거의 메세지를 가져오는 방법은 최신 메세지부터 과거의 메세지 순서로 데이터를 가져오게 된다. 초기 화면에서 가장 최신의 데이터를 보여주고 스크롤을 통하여 위로 올릴 수록 과거의 메세지가 나오게 된다. 이러한 메신저의 특성을 이용하여 효율적으로 메세지를 저장하고 가져오도록 설계하였다. 우선 채팅방별 특정 시간, 개수별 메세지를 Redis 에 저장한다. 그 이후 개수 혹은 시간 단위로 묶인 덩어리를 NoSQL 로 저장하게 된다. Redis 에는 해당 채팅방의 가장 최신화된 덩어리의 Key 값을 저장한다.

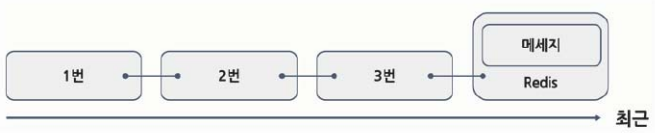

그림 5. STOMP 를 활용한 초대 메세지 수신

그림 5 와 같은 형태의 저장 구조가 발생하게 된다. 1-3 번은 NoSQL 속에 저장되어 있으며 각 메세지 덩어리가 이전의 메세지 덩어리의 Key 를 갖는다. Redis 에 남아있는 메세지는 아직 NoSQL 에 저장되지 않은 메세지이며 가장 가까운 과거의 메세지 덩어리에 대한 NoSQL Key 를 가진다. 이러한 구조의 경우 Redis 에 메세지가 빠르게 저장되며 과거의 데이터를 불러올 때에는 Redis 의 잔여 메세지를 가져온 후 그 전 덩어리를 불러오게 된다. 또한 스크롤을 통하여 이전 메세지를 더 가져오게 된다면 가져온 NoSQL 의 메세지 덩어리에서 이전 덩어리에 바로 접근하여 데이터를 가져오게 되므로 RDB 에서의 복잡한 쿼리보다 훨씬 빠른 속도를 보여줄 수 있다.

3-4. HAProxy 를 활용한 Scale Out

위와 같이 분산 설계된 서버들을 Scale Out 하기 위하여 HAProxy 를 활용하도록 설계하였다. 특정 기능의 서버에 부하가 생길 경우 해당 서버를 증설하여 HAProxy 를 활용하여 Load Balancer 의 역할을 수행할 수 있다.

그림 6 과 같이 부하가 생기는 서버를 증설하고 HAProxy 의 Load Balancing 기능을 활용하여 쉽게 Scale Out 이 가능하다. 채팅 서버 증설의 경우 Kafka 와 채팅 처리 서버의 증설을 같이 진행하여 확장할 수 있다.

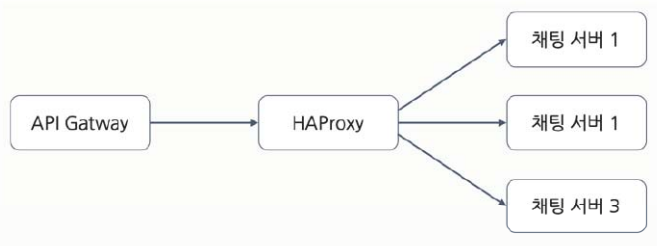

그림 6. 서버 증설 시 구조

4. 결론

본론에서 대용량 트래픽 처리를 위해 분산서버를 이용하여 채팅 메세지를 송수신하는 방법과 서버를 증설하는 방법을 분석하였다. 마이크로서비스 아키텍처는 모놀리식 애플리케이션에 비해 높은 응집도(Cohesion)와 낮은 결합력(Coupling)을 갖는다. 또한 유지보수에 있어서 Scale Out 에 용이하다. 향후 개발될 웹 서비스에 변화 트래픽에 적절히 대응하는 구조를 적용할 경우 성능 향상이 기대된다.

참고문헌

[1] Search Reliability Engineering [Online]. Available : https://deview.kr/2018/schedule/253
[2]KHIN ME ME THEIN, Apache Kafka: Next Generation Distributed Messaging System, ISSN, Vol.03, Issue.47, pp. 9478-9483, 201
[3]Park, Joon Seok, Design and Implementation of WebApplication Framework for Classroom Management using REDIS, 2014
[4]Daniel E. Eisenbud et al., Maglev: a fast and reliable software network load balancer, In Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI`16) , pp.523-535, Mar. 2016

Acknowledgement

[본 논문은 과학기술정보통신부
정보통신창의인재양성사업의 지원을 통해 수행한
ITC 멘토링 프로젝트 결과물입니다.]