

碩士學位論文

EventSourcing과 CQRS를 적용한 IoT
시스템에서 사용자에게 분석의 효율성을
제공하기 위한 폴리글랏 구조 설계

**Polyglot structure design to provide users
with the efficiency of analysis within IoT
systems with EventSourcing and CQRS**

한밭대학교 情報通信專門大學院

모바일融合工學科

朴 民 煥

2021년 8월

EventSourcing과 CQRS를 적용한 IoT 시스템에서 사용자에게 분석의 효율성을 제공하기 위한 폴리글랏 구조 설계

**Polyglot structure design to provide users with the efficiency of
analysis within IoT systems with EventSourcing and CQRS**

指導教授 박 현 주

이 論文을 工學碩士學位
請求論文으로 제출함


2021년 5월


한밭대학교 情報通信專門大學院


모바일融合工學科

朴 民 煥

朴民煥의 碩士學位 論文을 認准함

審査委員長 이 창석 

審査委員 이 동호 

審査委員 박 현주 

2021년 6월

한밭대학교 情報通信專門大學院

목 차

그림목차	iv
국문요약	vi
I. 서론	1
1.1 도입	1
1.2 문제진술	1
1.3 목표	2
1.4 배치	3
II. 개념 및 시스템 설계	4
2.1 Command Query Responsibility Segregation	4
2.1.1 Command-Side	5
2.1.2 Aggregate	5
2.1.3 Query-Side	6
2.1.4 CQRS와 최종 일관성	7
2.1.5 CQRS의 장단점	8
2.2 Event Sourcing	8
2.2.1 Event Store	8
2.2.2 Event Sourcing의 장단점	9
2.2.3 Snapshot	9
2.3 Microservice Architecture	10
2.4 API Gateway	11
2.5 Service Discovery	12
2.5.1 The Client-Side Discovery Pattern	13

2.5.2 The Server-Side Discovery Pattern	14
2.6 Message Queue	15
2.7 Wisoft IoT Platform	16
III. 구현 기술 선택	17
3.1 Framework	17
3.1.1 Axon Framework	17
3.1.2 Axon Server	17
3.2 Databases	18
3.2.1 PostgreSQL	18
3.2.2 MongoDB	19
3.3 Message Queue	19
3.3.1 Kafka	19
3.4 API Gateway	19
3.4.1 Netflix Zuul	19
3.4.2 Netflix Eureka	20
IV. 구현	21
4.1 Platform Architecture	21
4.2 Command-Side Architecture	22
4.2.1 Command, Event	23
4.2.2 Command DTO	25
4.2.3 Event Store Metadata	26
4.3 Query-Side Architecture	27
4.3.1 PostgreSQL Projection	28
4.3.2 mongoDB Projection	30

V. 검증 및 평가	32
5.1 구현 요약	32
5.2 실험 결과	32
5.2.1 Postgresql Projection만을 연결하였을 때의 최종 일관성	33
5.2.2 MongoDB Projection만을 연결하였을 때의 최종 일관성	33
5.2.3 Projection을 동시에 연결하였을 때의 최종 일관성	34
5.2.4 실험 결과	35
VII. 결론 및 향후 연구	37
VIII. 참고 문헌	38
ABSTRACT	40

그림 목 차

그림 1 전 세계 IoT와 Non-IoT 연결 수 전망	1
그림 2 CQRS 패턴	4
그림 3 Command-Side	5
그림 4 주문 애그리거트	6
그림 5 Query-Side	6
그림 6 EventSourcing의 상태 변경	8
그림 7 Snapshot이 적용하지 않은 모델 Snapshot이 적용된 모델	9
그림 8 모놀리식 아키텍처와 마이크로서비스 아키텍처	10
그림 9 API Gateway	12
그림 10 Client-Side Discovery Pattern	13
그림 11 Server-Side Discovery Pattern	14
그림 12 Message Queue	15
그림 13 Wisoft IoT Platform	16
그림 14 Axon Framework	18
그림 15 IoT Platform Architecture	21
그림 16 Sensing Aggregate CQRS 구조	22
그림 17 Command-Side Architecture	23
그림 18 SensingCreationCommand	24
그림 19 SensingCreationEvent	24
그림 20 SensingCreationCommand DTO	25
그림 21 EventStore Metadata	26
그림 22 QuerySide Architecture	27
그림 23 Sensing Table	28

그림 24 PostgreSQL Projection Create에 대한 시퀀스 다이어그램	29
그림 25 Sensing Collection Data	30
그림 26 Sensing Data CSV	30
그림 27 MongoDB Projection Create에 대한 시퀀스 다이어그램	31
그림 28 PostgreSQL Projection 최종 일관성 도달 테스트	33
그림 29 MongoDB Projection 최종 일관성 도달 테스트	34
그림 30 동시적으로 요청 시 최종 일관성 도달 테스트	34
그림 31 실험 1과 실험 3의 RDBMS 비교	35
그림 32 실험 2와 실험 3의 NoSQL 비교	35

국 문 요 약

EventSourcing과 CQRS를 적용한 IoT 시스템에서 사용자에게 분석의 효율성을 제공하기 위한 폴리글랏 구조 설계

논문 제출자 박 민 영

지도 교수 박 현 주

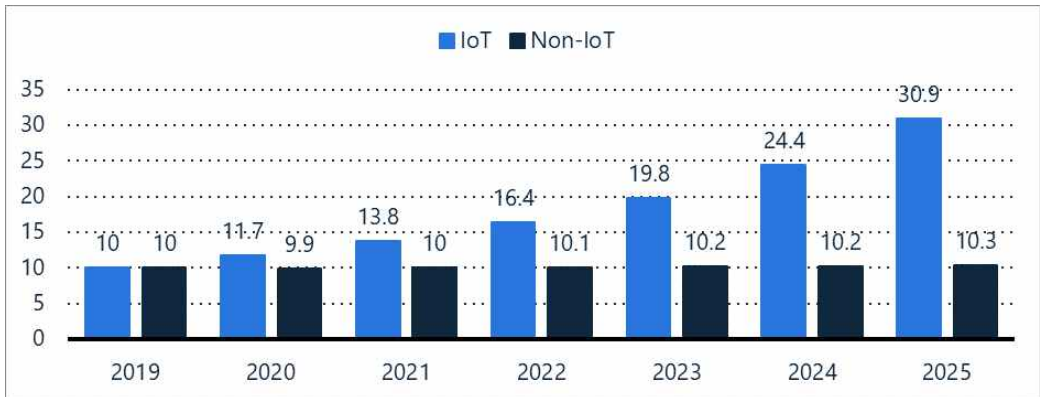
IoT 기술이 점차적으로 발전하면서 연결되는 장치들이 증가함에 따라 데이터의 양이 지속적으로 증가하고 있다. 결국 서비스가 지속됨에 따라 부하의 양도 늘어나고 있는 추세이다. 또한 대부분의 시스템은 정형화된 아키텍처로 구성되어 있는 경우가 많기 때문에 많은 양의 데이터를 수집한다 하더라도 질 좋은 분석 결과를 얻기가 힘든 상황이다. 부하 분산을 줄이고 대용량 데이터를 효율적으로 제공하기 위해 본 논문에서는 대용량 데이터 환경에 맞는 아키텍처를 보여준다. 명령, 조회에 대한 책임을 분리하는 CQRS 패턴과 장치에서 발생하는 이벤트 로그를 모두 저장하는 패턴인 Event Sourcing을 사용한 아키텍처로 변경한다. 데이터베이스는 MSA로 서비스를 나누어 폴리글랏 구조를 만들어 쿼리를 활용하여 조회할 수 있는 데이터는 관계형 데이

터베이스인 PostgreSQL를 사용한다. 분석에 활용할 데이터는 NoSQL인 MongoDB를 활용하여 사용자에게 제공한다. 제안하는 플랫폼은 다양한 환경에서의 대용량 데이터를 질적이면서도 효율적으로 데이터 분석에 활용하기 위함이다.

I. 서론

1.1 도입

IoT(Internet Of Things) 기술이 발전하면서 연결되는 IoT 장치들의 종류가 다양해지면서 수집되는 데이터의 양도 지속해서 증가하고 있고 다양해지고 있다. IoT Analytics에 따르면, 전 세계 사물인터넷(IoT) 장치는 2025년까지 309억 대로 예상되며, 2021년의 138억 대보다 2배 이상 늘어날 전망이다. IoT의 예로는 스마트 홈·카·팩토리 등이, non-IoT에는 스마트폰·노트북·컴퓨터 등이 있다. 또한 2025년 IoT 장치는 non-IoT 장치(100억 대)의 3배 수준에 이를 것이다[1].



출처: IoT Analytics, 테크월드 재가공

그림 1 전 세계 IoT와 Non-IoT 연결 수 전망

1.2 문제 진술

IoT 애플리케이션 사물은 기록 분석을 위해 데이터를 저장하는 경우가 많다. 그러나 대부분의 시스템은 정형화된 아키텍처로 구성되어 있기 때문에 과정의 데이터 없이 최종 데이터만을 데이터 저장소에 저장을 한다. 결과적으로 다양한 장치들이 데이터를 측정하고 데이터를 저장한다 하더라도 분석하고 실생활에 활용하기에는 한계점이 존재한다. 예를 들어 사물이 주위에 어떠한 환경에 영향을 받아 측정한 결과값이 나오는지의 여부이다. 또 다른

문제점은 서비스에 장치들의 등록이 많아지면서 부하도 증가하여 성능의 저하로 이어진다. 결국 비용 문제, 성능 문제 등의 문제들이 발생을 하게 된다. 본 논문에서 답하고자 하는 것은 어떻게 서비스의 부하를 줄일 것이며 또한 데이터를 효과적으로 분석하기 위한 방법이 무엇인지에 대한 것이다.

1.3 목표

대용량 데이터를 효율적으로 제공하고 다양한 데이터를 효과적으로 활용하기 위해 본 논문에서는 대용량 데이터 환경에 맞는 아키텍처를 보여준다. CQRS(Command Query Responsibility Segregation) 패턴을 사용하여 쓰기 모델, 읽기 모델로 나누고 장치에서 발생하는 모든 데이터를 이벤트로 저장하기 위해 EventSourcing 패턴을 사용한다. EventSourcing 패턴은 사용자가 시스템에서 수행하는 도메인별의 작업을 나타내는 이벤트 로그이며 이벤트 저장소에 시계열 데이터를 저장하기 때문에 과거 시점을 전달받아 분석을 위한 데이터로 유용하게 활용할 수 있다. 이벤트가 발생하면 읽기 모델에 업데이트를 해야 한다. 쓰기 모델의 이벤트 저장소와 읽기 모델의 데이터 저장소의 일관성을 도달하기 위해 MessageQueue인 Kafka를 사용하여 대용량 데이터를 빠르게 전달을 할 수 있게 한다. 또한 읽기 모델의 데이터 저장소는 MSA(Microservice Architecture)를 이용하여 폴리글랏 구조로 변경을 한다. 폴리글랏 구조의 장점은 사용자가 원하는 데이터를 빠르게 전달하기 위해 필요한 데이터 저장소는 추가로 제공할 수 있다. 데이터 저장소는 대표적으로 많이 사용하는 RDBMS, NoSQL을 활용한다. 조인에 강점이 있는 RDBMS는 그에 맞는 데이터를 조인해 최종 데이터를 사용자에게 보여주고 분석에 활용하기 위한 데이터는 NoSQL을 이용하여 비정형 데이터를 이벤트 저장소에서 시계열로 전달받아 CSV 파일 등으로 출력하여 제공하도록 한다. 결과적으로 사용자는 각 데이터 저장소에서 쿼리를 보내 적절한 데이터를 받아 활용할 수 있다.

1.4 배치

본 논문에서 위와 같은 내용을 전달하기 위한 구성은 다음과 같다. 먼저 2장에서는 명령 쿼리 책임 분리 개념 및 설계와 이벤트 소싱, 마이크서비스, 서비스 디스커버리, 메시지 큐에 대한 개념을 설명한다. 3장은 선택한 기술 및 정의에 대한 설명을 하고 4장에서는 이 논문의 핵심 구현을 설명한다. 5장에서는 구현한 플랫폼의 성능을 설명하고 6장에서는 결론을 짓고 향후 연구에 대한 내용을 기술한다.

II. 개념 및 시스템 설계

이 장에서는 논문의 기초를 설정하는 용어, 개념 및 시스템 설계를 다룬다.

2.1 Command Query Responsibility Segregation

CQRS 패턴은 두 개의 독립적인 하위 시스템 쓰기 모델과 읽기 모델을 설명하며, 사용자 및 사물 인터넷이 변경하면 쓰기 모델이 업데이트된다. 그런 다음 시스템은 쓰기 데이터 모델을 처리하여 쿼리 모델을 만든다.

쓰기 모델은 IoT 장치 등 실생활에서 수집된 시계열 데이터를 이벤트 저장소에 저장하고 저장된 데이터를 적절하게 분석한 후 해당 데이터를 읽기 모델 측 데이터 저장소에 투영한다[2, 3]. 그림 2는 CQRS 패턴의 간략한 그림을 나타낸다.

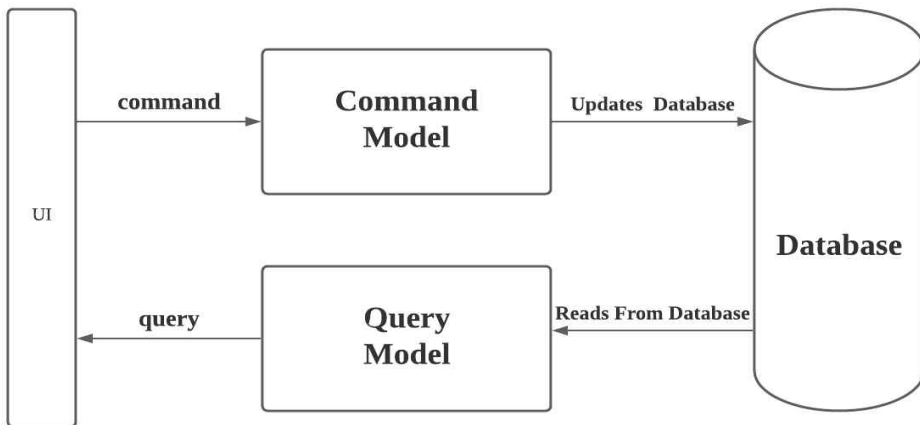


그림 2 CQRS 패턴

사용자가 변경사항을 만든다면, 변경사항의 데이터는 command를 만들어 Command Model로 전달되고 데이터베이스에 데이터를 저장한다. 변경에 대한 결과는 Query Model로 전달되어 사용자에게 제공된다.

2.1.1. Command-Side

의도를 표현하기 위한 명령을 Command Model에 전달한다. IoT 장치에서 보낸 단순한 시계열 데이터를 전달할 수 있고 장치의 상태 변경을 위한 명령일 수도 있다. 예를 들어 화재와 같은 위급상황이 발생했을 시 가스 센서가 가스의 유/무를 측정을 하여 불이 더 번지지 않게 하기 위해 주위 셔터를 내리는 것과 같이 활용할 수 있다. 명령을 전달할 때 Command Handler가 데이터 형태에 맞는 Aggregate에 전달하여 의사를 표현을 하기 위한 명령을 Event Store에 투영한다. 그림 3은 Command-Side에 대한 간략한 그림을 보여준다.

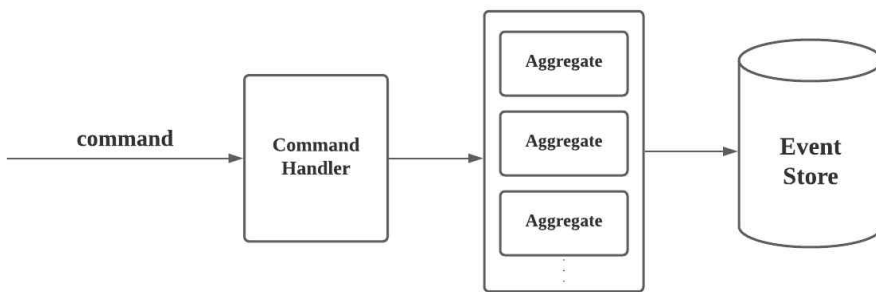


그림 4 Command-Side

2.1.2. Aggregate

애그리거트는 DDD(도메인 기반 설계의 패턴)이다. 단일 단위로 처리할 수 있는 도메인 개체이며 한 예시는 주문과 그 라인업(배송지 정보, 주문자, 주문 목록 등)이다. 다음과 같이 하위 모델로 구성되는데 이때 이 하위 개념을 표현한 모델을 하나로 묶어서 ‘주문’이라는 상위 개념으로 표현할 수 있다. 많은 객체들을 애그리거트로 묶어서 보면 더 상위 수준에서 도메인 간의 관계를 파악할 수 있다[3]. 그림 4는 주문 애그리거트의 예시를 나타낸다.

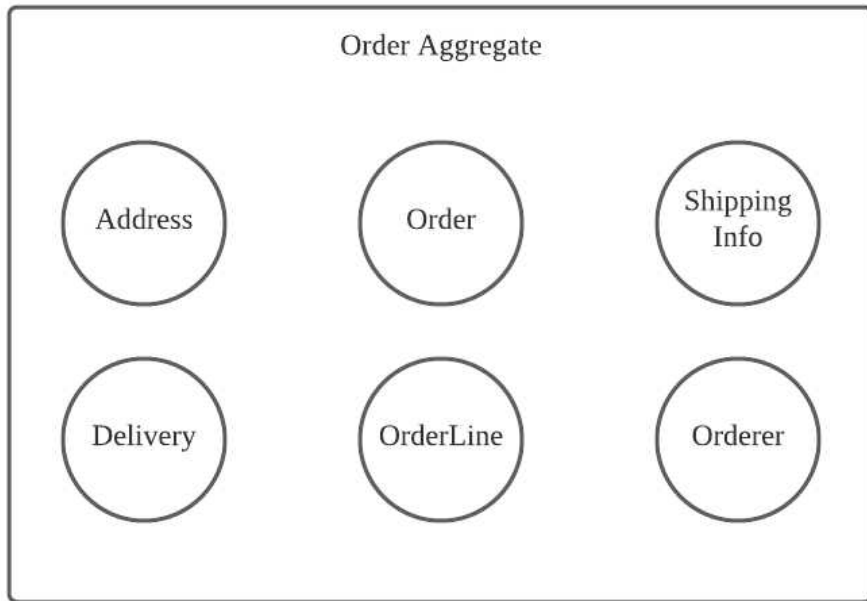


그림 5 주문 애그리거트

2.1.3. Query-Side

Query-Side는 데이터를 조회하는 방법만이 포함되어 있다. Command-Side에서 Event를 받아 RDBMS, NoSQL 등 데이터베이스에 데이터를 투영한다. 사용자는 Query-Side 데이터저장소를 이용하여 적절한 Query를 제공받는다. 그림 5는 Query-Side에 간략한 그림을 보여준다.

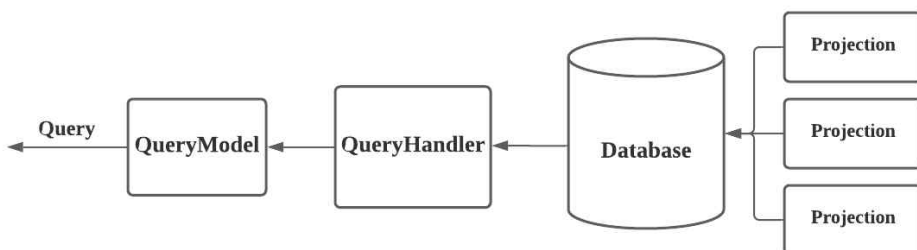


그림 6 Query-Side

그림과 같이 여러 개의 Projection이 하나 이상의 데이터베이스에 저장할 수 있으며 데이터베이스에 저장된 데이터들이 QueryHandler를 거쳐 QueryModel을 만들어 제공을 한다.

2.1.4. CQRS와 최종 일관성

CAP 이론(Consistent, Available, Partition-Tolerant)은 분산 시스템은 아래의 세 가지를 동시에 만족할 수 없음을 증명하는 이론이다[4].

일관성: 일관성은 연결하는 노드에 관계없이 모든 클라이언트가 동시에 동일한 데이터를 볼 수 있음을 의미한다. 이를 위해서는 데이터가 하나의 노드에 기록될 때마다 시스템의 관련된 노드에 즉시 전달되거나 복제되어야 한다.

가용성: 모든 사용자들의 요청과 응답이 가능해야 하며, 몇몇 노드의 장애가 발생하여도 정상적인 노드에 영향을 미치지 않아 된다.

파티션 허용 오차: 네트워크에 일시적인 통신 장애가 있는 경우에도 시스템의 두 노드가 진행할 수 있는 기능이다. 파티션 허용 범위가 높을수록 데이터베이스는 클라이언트가 더 많이 사용할 수 있지만 액세스 가능한 데이터는 일관성이 떨어지며 그 반대의 경우도 마찬가지이다.

분산 시스템에서, 데이터 일관성(data consistency), 시스템 가용성(system availability), 네트워크 분할 허용성(network to partition-tolerance) 간의 고유한 균형이 존재한다. 분산 시스템은 이들 세 특성 가운데 두 가지를 제공할 수 있지만, 세 가지 모두는 제공하지 않는다.

CQRS 패턴의 구현은 분산 시스템을 구현하는 것이다. 분산 시스템은 대부분 분산 데이터베이스를 사용하기 때문에 필히 만족해야 하는 두 가지는 가용성 및 파티션 허용 오차이다. CQRS 패턴은 다음의 두 가지는 필히 만족하고 일관성도 데이터의 오차가 존재하더라도 결국에는 도달하기 때문에 이 부분도 만족하여 분산시스템에 적합한 패턴이라고 말할 수 있다.

2.1.5. CQRS의 장단점

읽기 모델과 쓰기 모델이 분리되어 있기 때문에 데이터베이스의 읽기 및 쓰기를 동시에 하지 않아 분리되지 않은 모델에 비해 성능이 저하되지 않는다. 이벤트 스토어에 현재 상태의 데이터만 저장하는 것이 아니라 각각의 이벤트를 저장하고 있기 때문에 이벤트에 따른 데이터 변화를 쉽게 알 수 있으며 읽기 모델에서 현재 상태의 데이터를 언제든지 다시 복구할 수 있다. 또한 이벤트를 활용함으로써 개체의 특정 패턴 분석 등으로도 활용할 수 있고 쓰기 모델에서 활용 용도에 따른 데이터베이스를 선택할 수 있다. 그러나 쓰기 모델과 읽기 모델의 데이터 저장소가 일관성을 계속 도달하기 위해 쓰기 모델의 이벤트 스토어와의 I/O 작업이 계속 일어난다.

2.2 Event Sourcing

이벤트 소싱(Event Sourcing)이란 현재의 상태를 저장하는 전통적인 아키텍처와 달리 애플리케이션 또한 장치에서 발생하는 이벤트 로그(상태를 변경하는 데이터)를 모두 저장하는 아키텍처 패턴이다. 그림 6은 이벤트 소싱의 상태 변경 예시를 나타낸다.

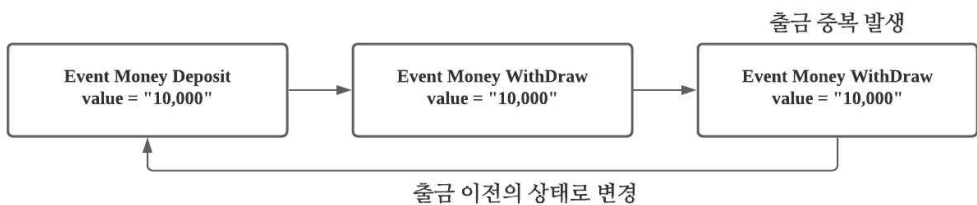


그림 7 EventSourcing의 상태 변경

2.2.1 Event Store

이벤트 스토어는 이벤트를 저장하기 위한 데이터베이스이다. 한번 들어온 이벤트는 절대 변경할 수 없으며 오류 이벤트가 들어오더라도 하나의 이벤

트로 인식하여 그것 또한 변경할 수 없다. 특정 유형의 이벤트 스트림을 구별할 수 있고 식별자로 고유하게 식별한다.

2.2.2 Event Sourcing의 장단점

이벤트 소싱의 장점은 위에 말한 것처럼 데이터의 상태 변경을 모두 이벤트로 저장하기 때문에 오류가 발생하거나 특정 상태의 값을 알고자 할 때 해당 시점까지 이벤트를 재생하여 상태를 복원할 수 있다. 이벤트의 변경(수정 및 삭제)을 하지 않아 불변성을 유지한다. 단점은 읽기 모델에서 이벤트 저장소에 저장된 이벤트 스트림을 모두 읽어야 한다. 이벤트의 개수가 적을 때는 문제가 되지 않을 수 있지만 이벤트의 개수가 늘어날수록 현재 상태를 복원하기까지 많은 시간이 소요될 것이다.

2.2.3 Snapshot

현재 상태를 복원하기에 빠르면서 가장 일반적인 방법은 스냅샷을 사용하는 것이다. 읽기 모델에서 이벤트를 저장할 때 일정량 이상 받았을 경우 스냅샷을 찍는다. 다시 현재 상태를 재생하고자 하면 읽기 모델에서 스냅샷을 찍은 이후의 모든 이벤트를 재생하여 이전에 말했던 시간 소요 부분을 어느 정도 해소할 수 있다. 그림 7은 (1) Snapshot이 적용되지 않은 Event Stream과 (2) Snapshot이 적용된 Event Stream을 비교한 것이다.

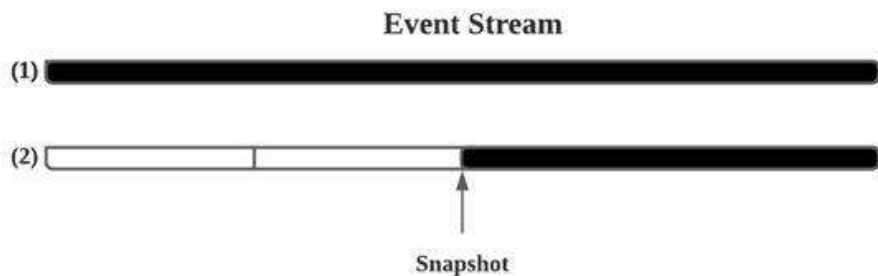


그림 8 Snapshot이 적용하지 않은 모델 Snapshot이 적용된 모델

2.3 Microservice Architecture

마이크로서비스 아키텍처가 나온 배경은 모놀리식(Monolithic) 아키텍처의

문제점 때문이다. 먼저 모놀리식 아키텍처는 구현 뷰를 단일 컴포넌트(하나의 실행파일, WAR)로 구성되어 있기 때문에 배포가 쉽고 관리 포인트가 적다는 장점이 있지만 하나의 서비스만 문제가 생겨도 장애의 전파가 쉽고 시스템 확장의 비용도 많이 들기 때문에 개발하기에 앞서 점차 늘어나는 대용량 데이터의 처리에 한계에 봉착했다. 그 문제를 해결하고자 마이크로서비스가 등장하였다. 마이크로서비스 아키텍처는 하나의 애플리케이션을 각각의 역할을 가진 작은 애플리케이션으로 분리하는 서비스 지향 아키텍처 접근 방식이다. 그림 9의 모놀리식과 마이크로서비스의 비교를 나타낸다.

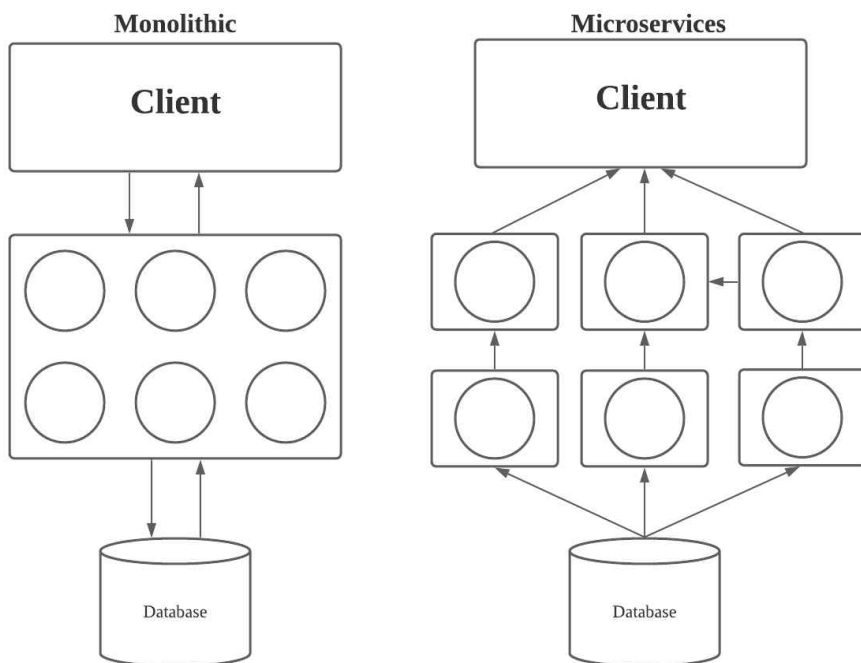


그림 9 모놀리식 아키텍처와 마이크로서비스 아키텍처

마이크로서비스는 다음과 같은 특징이 있다.

모듈성: 마이크로서비스 아키텍처는 서비스를 모듈성의 단위로 사용한다. 각 서비스가 함부로 규칙을 어기고 호출할 수 없게 만드는 API라는 경계선이 존재하다. 따라서 다른 서비스 API 우회하여 그 내부 클래스에 맘대로 들

어오지 못한다.

서비스 별 데이터베이스 : 마이크로서비스는 서로 느슨하게 결합되어 오직 API를 통해서만 호출한다. 때문에 각각 자체 데이터베이스를 갖고 있다. 런타임에 서비스는 완전히 분리되어 있기 때문에, 다른 서비스가 데이터베이스 락을 획득해 내 서비스를 블로킹하는 일은 일어나지 않는다.

느슨한 결합: 느슨하게 결합된 서비스도 마이크로서비스 아키텍처의 주요 특성 중 하나이다. 앞서 말한 것처럼 서비스는 구현 코드로 감싼 API를 통해서만 상호 작용하므로 서비스 외부로는 영향을 끼치지 않고 서비스 구현 코드를 바꿀 수 있다. 그럼으로써 유지 보수성, 테스트 성을 높이고 애플리케이션 개발 시간을 단축하는 효과가 있다.

네트워크 지연: 네트워크 지연은 분산 시스템의 고질적인 문제이다. 서비스를 여러 개로 나누면 서비스 간 왕복 횟수가 늘어난다. 그렇기 때문에 배치 API를 구현하거나 IPC를 메서드나 함수 호출로 대체하는 등의 식으로 서비스 결합에 따른 지연시간을 줄인다.

동기 IPC로 인한 가용성 저하: REST API를 동기 호출하는 것이 가장 쉬운 구현 방법이지만, 타 서비스 중 하나라도 불능이 경우 서비스되지 않기 때문에 REST와 같은 동기 프로토콜 대신 비동기 메시징으로 강한 결합도를 제거하여 가용성을 높인다.

2.4 API Gateway

MSA는 작은 애플리케이션을 개발 및 운영하는 아키텍처이다. 서비스를 잘게 나누면서 작게는 수십 개에서 많게는 수백 개까지 서비스가 나뉘게 된다. 이를 직접적으로 호출하게 되면 각 서비스마다 IP가 다르기 때문에 각각의 IP를 관리하기 힘들 뿐 아니라 클라우드 환경에서는 동적으로 변하기도 하여 관리의 어려움이 있다. 또한 서비스마다의 인증/인가 절차를 구현해야 하는 등의 문제점이 존재한다. 이러한 문제점을 API Gateway로 인해 해결할 수가 있다. 아래의 그림을 보면 알 수 있듯이 API Gateway를 사용하지 않은 모델은 Client가 각각의 서비스의 주소를 알고 호출해야 하지만 API Gateway

y를 적용한 모델을 살펴보면 Client는 API Gateway의 주소만을 알고 있어도 API Gateway가 내부 서비스의 API를 적절하게 호출하게 된다. 그림 9는 API Gateway 사용의 장점을 나타낸다.

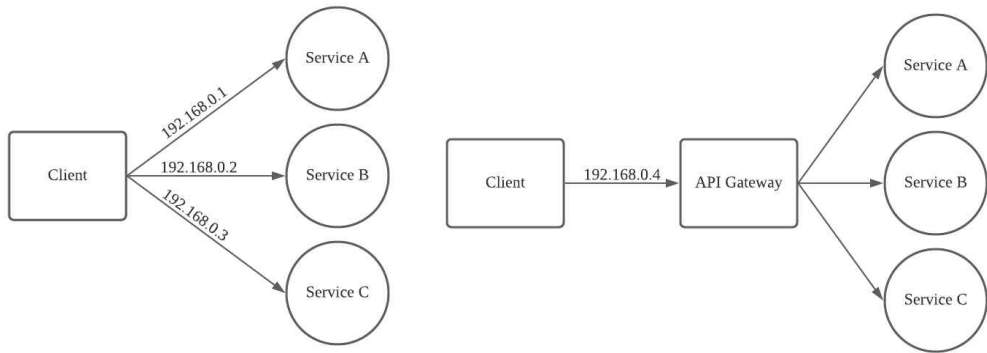


그림 10 API Gateway

API Gateway의 특징은 다음과 같다.

인증 및 인가 (Authentication and Authorization): 각 서비스마다의 인증/인가(인증서 관리, SSL)를 서버 제일 앞단에 있는 API Gateway가 처리해 줌으로써 중복과 복잡성을 해결한다.

로드밸런싱(load balancing): API Gateway에 요청을 함으로써 서비스에 대한 부하 분산을 적절하게 하여 부하를 줄일 수 있다.

모니터링(monitoring): 서비스의 부하 및 분석하고 싶을 경우 모니터링 툴을 추가할 수 있다.

2.5 Service Discovery

서비스 디스커버리를 설명하기에 앞서 서비스는 API Gateway를 통해 API를 호출한다고 했다. 하지만 REST API 호출을 하기 위해 API Gateway도 요청하고자 하는 서비스의 네트워크 위치(IP, Port)를 알아야 한다. 기술의 발

전으로 대부분의 시스템은 클라우드 상에 올라가서 동적으로 네트워크가 부여된다. 때문에 서비스의 네트워크를 관리할 수 있는 시스템 도입이 필요하다.

서비스 디스커버리는 위와 같은 문제점을 해결해 주는 정교한 서비스 검색 메커니즘을 이용한다[5].

아래는 서비스 디스커버리의 두 가지 주요 서비스 검색 패턴을 설명한다.

2.5.1 The Client-Side Discovery Pattern

클라이언트 측 검색 패턴은 먼저 각각의 서비스가 실행이 되면 데이터베이스에 서비스의 네트워크 위치를 서비스 레지스트리에 보낸다. 서비스 레지스트리는 서비스들의 위치 정보를 데이터베이스에 저장을 한다. 클라이언트는 사용 가능한 서비스 중 요청하고자 하는 서비스에 대해 Query를 하여 네트워크 위치 정보를 얻고 요청하고자 하는 서비스 API에 요청을 한다. 아래의 그림은 이 패턴의 흐름을 보여준다. 그림 10은 클라이언트 측 검색 패턴의 예를 나타낸다.

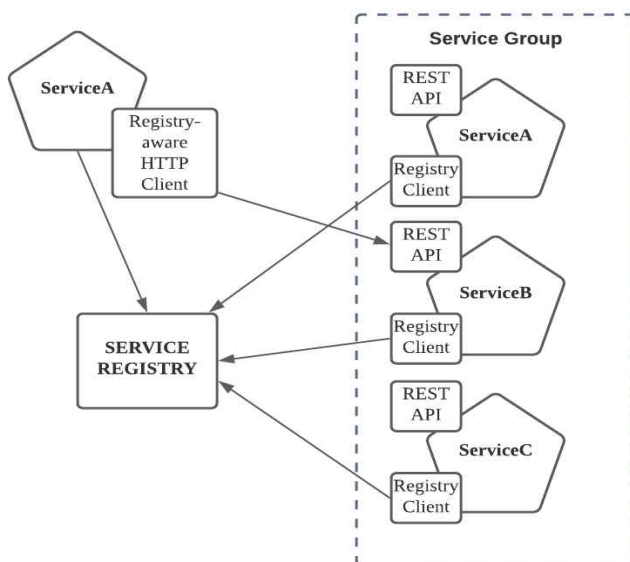


그림 10 Client-Side Discovery Pattern

이 패턴의 장점은 관리 포인트가 서비스 레지스트리밖에 없어 비교적 간

단하고 클라이언트가 사용 가능한 서비스 네트워크 위치를 알고 있기 때문에 지능적으로 로드밸런싱을 구성할 수 있다. 그러나 클라이언트와 서비스 레지스트리와의 의존도가 높아지며 Client마다 찾고자 하는 서비스에 대한 로직을 구현해야 하는 단점이 존재한다.

2.5.2 The Server-Side Discovery Pattern

서버 측 검색 패턴은 서비스가 실행이 되면 데이터베이스에 서비스의 네트워크 위치를 서비스 레지스트리에 보낸다는 점은 같으며 클라이언트가 로드밸런서를 이용해 각각의 서비스를 찾는다는 점이 다르다. 클라이언트가 로드밸런서에 요청을 보내면 로드밸런서가 서비스 레지스트리에 Query 명령을 보내 API를 호출하고자 하는 서비스의 네트워크 위치를 찾고 요청을 보낸다. 그림 11은 서버 측 검색 패턴의 예를 나타낸다.

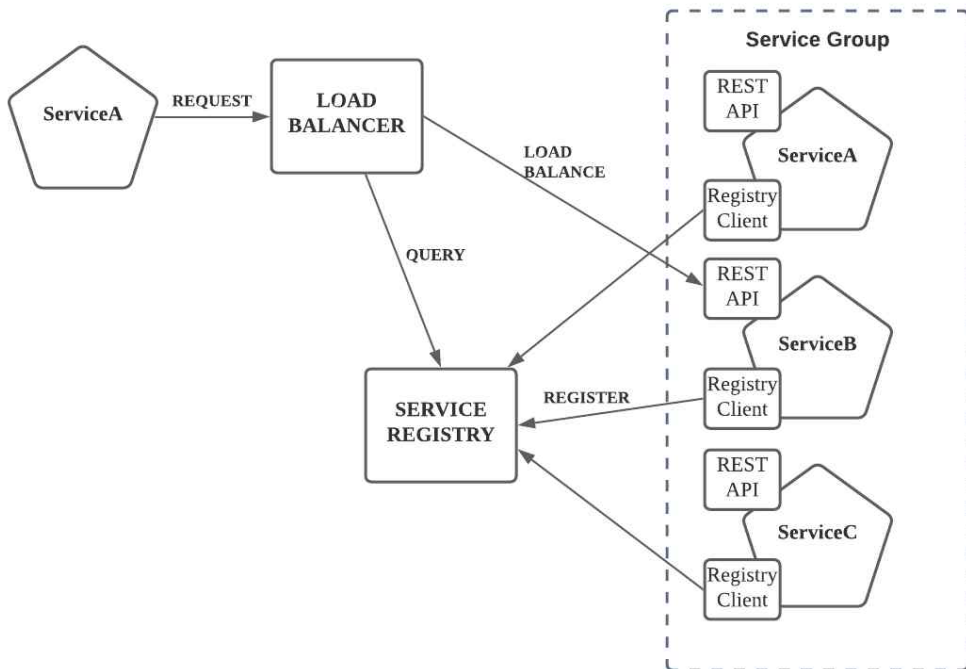


그림 11 Server-Side Discovery Pattern

2.6 Message Queue

메시지 지향 미들웨어(Message Oriented Middleware: MOM)는 다른 애플리케이션이나 서비스 사이에서 비동기 메시지를 사용해서 데이터 송수신을 의미한다. MOM을 구현한 시스템을 메시지 큐(Message Queue: MQ)란 프로세스 간의 데이터를 교환할 때 사용하는 통신 방법 중 하나이다. 메시지 큐는 두 개 이상의 프로세스 / 스레드 간에 비동기 통신 패턴을 구현하기 때문에 송신자와 수신자가 동시에 메시지 큐와 상호 작용할 필요가 없다. 일반적으로 브로커를 구성하고 명명된 메시지 큐를 정의한다. 송신자가 메시지를 보내면 브로커를 통해 수신자에게 전달되는 형태이고 큐를 활용하기 때문에 애플리케이션이 연결되기 전이거나 끊기더라도 데이터를 큐에 저장하여 애플리케이션이 활성화되었을 때 다시 데이터를 보낼 수 있다. 대표적으로 애플리케이션 및 서비스 사이에서 메시지 교환할 때 Message Queue의 오픈소스에 기반을 둔 표준 프로토콜인 AMQP(Advanced Message Queuing Protocol)을 이용하고 Java Application은 메시지 통신을 위한 자바 메시지 기반 미들웨어인 JMS(Java Message Service)를 이용하여 통신을 한다. 그림 12는 Message Queue를 보여준다.

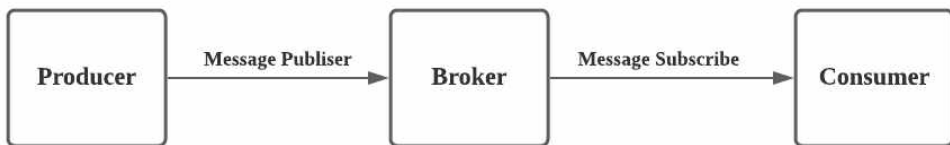


그림 13 Message Queue

메시지 큐의 장점은 다음과 같다.

비동기(Asynchronous): 큐(Queue)를 이용하기 때문에 바로 데이터를 처리하지 않아도 된다.

안정성(stability): 중요한 데이터는 확실히 전달할 수 있다.

과잉(Redundancy): 실패할 경우 재실행이 가능하다.

확장성(Scalable): 다수의 프로세스들이 큐에 메시지를 보낼 수 있다.

탄력성(Resilience): 일부가 실패하여도 전체에 영향을 받지 않는다.

2.7 Wisoft IoT Platform

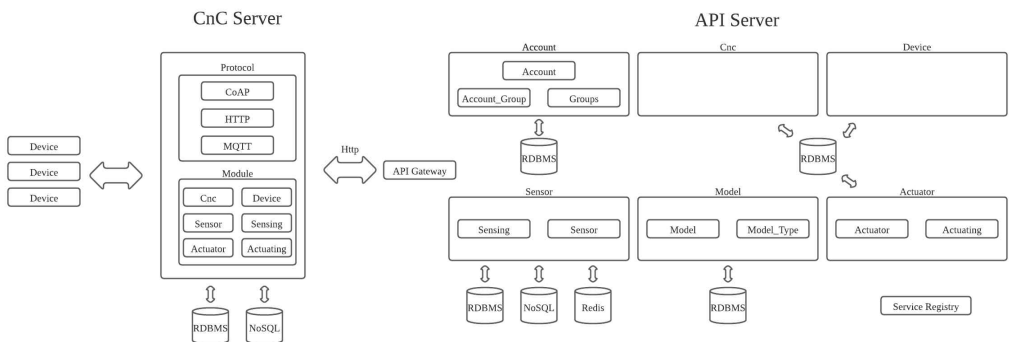


그림 14 Wisoft IoT Platform

WiSoft IoT Platform은 한밭대학교 무선통신소프트웨어 연구실에서 개발한 IoT Platform으로 표준 문제와 대용량 데이터 처리 문제 등을 해결하고자 개발하였다. Wisoft IoT Platform은 그림 13과 같이 구성되어 있다. CnC(Data Collection & Control Server) Server의 역할은 CoAP, HTTP, MQTT를 지원하는 선택적 프로토콜 기반의 실시간 데이터 수집이며 또한 Device와 API Server를 연결하는 Edge Server의 역할을 담당한다. API Server는 MSA의 구조로 되어 있으며 CnC Server에서 보낸 API 요청에 따라 API Gateway를 거쳐 데이터베이스에 데이터를 저장하고 센서를 제어한다. 본 논문에서는 API Server의 구조를 변환할 것이며 특히 Sensor Aggregate를 부하 분산 및 대용량 데이터 활용에 대한 효율적인 구조로 변환한다[7].

Ⅲ. 구현 기술 선택

3.1 Framework

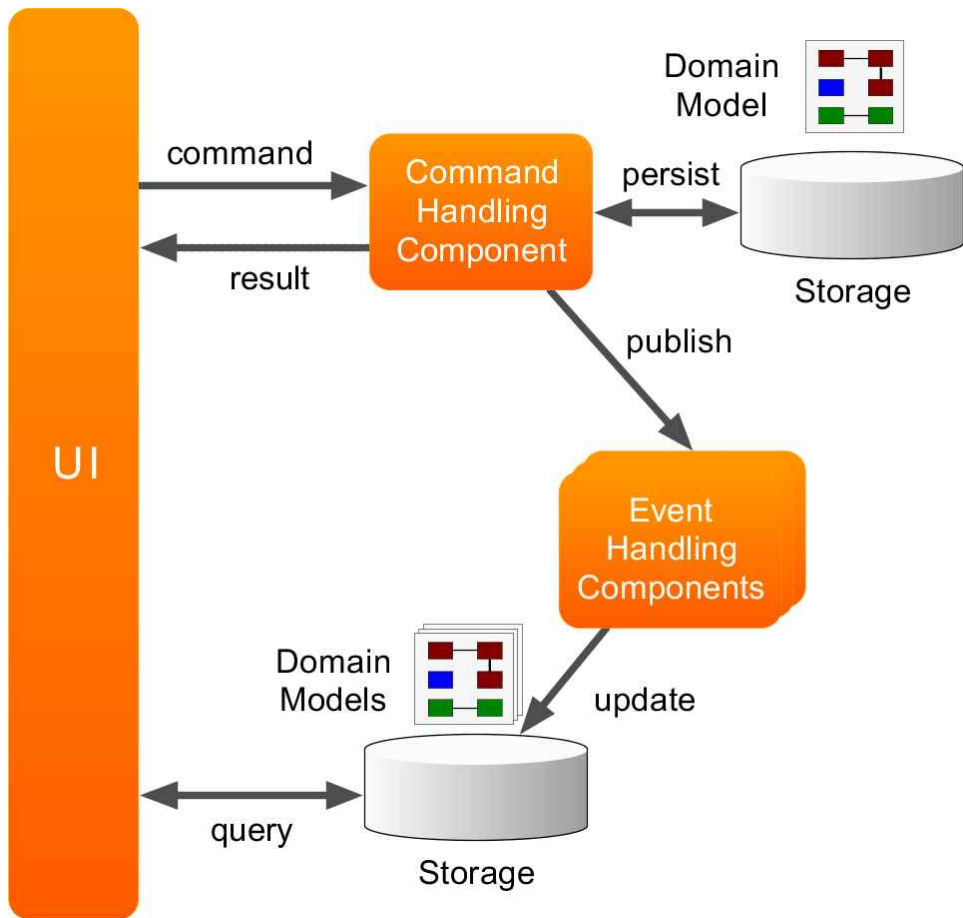
3.1.1 Axon Framework

Axon Framework는 2010년 최초로 프로젝트가 시작되었으며 네덜란드에서 설립된 AxonIQ에서 개발을 주도하고 있는 오픈소스이다. DDD, EventSourcing, CQRS를 주력으로 제공하고 있으며, Java 애플리케이션을 개발하는 통합적이고 생산적인 방법을 제공한다. Axon은 특히 미션 크리티컬 비즈니스 애플리케이션의 확장 및 배포를 위해 프로그래밍 모델에 대한 엔터프라이즈 준비 운영 지원을 제공하는 전문 인프라와 프로그래밍 모델을 모두 포함하고 있다. Axon Framework는 기본적으로 CQRS 구조를 따르고 있다. 사물이나 클라이언트에서 명령이 들어오면 CommandHandler를 통해 이벤트 스토어에 Event를 저장한다. 저장된 이벤트는 EventHandler를 통해서 읽기 모델의 데이터 저장소에 이벤트를 투영시킨다. 사용자는 읽기 모델의 데이터 저장소에서 Query를 하여 데이터를 제공받고 있다[8].

3.1.2 Axon Server

이벤트 스토어의 기준은 다음과 같습니다.

이벤트는 추가만 가능하고 입력 삭제, 수정이 불가능하다. 여러 이벤트가 하나의 트랜잭션 처리가 되어야 한다면, 트랜잭션 단위로 Commit 혹은 Rollback 되어야 한다. 또한 모든 이벤트는 삽입된 순서로 읽기가 가능해야 한다. Axon Server는 이 기준을 충족하며, 또한 빠른 시간 내에 이벤트 기반 데이터베이스를 구축할 수 있다. 전체적인 Axon Framework 구조는 그림 14와 같다[9].



출처 : <https://docs.axoniq.io/reference-guide/architecture-overview>

그림 15 Axon Framework

3.2 Databases

읽기 모델에서 사용할 데이터베이스는 다음과 같은 모델을 사용한다.

3.2.1 PostgreSQL

PostgreSQL은 복잡한 데이터 워크 로드를 안전하게 저장하고 확장하는 많

은 기능과 결합된 SQL 언어를 사용하고 확장하는 강력한 오픈 소스 객체 관계형 데이터베이스 시스템이다[10].

3.2.2 MongoDB

MongoDB는 크로스 플랫폼 도큐먼트 지향 NoSQL 데이터베이스 시스템이다. 전통적인 관계형 데이터베이스가 테이블을 이용한 조인이라면 NoSQL인 MongoDB는 일관성 모델을 이용하는 데이터의 저장 및 검색을 위한 메커니즘을 제공하며 JSON과 유사한 형식의 데이터를 저장한다[11].

3.3 Message Queue

3.3.1 Kafka

아파치 카프카(Apache Kafka)는 아파치 소프트웨어 재단이 스칼라로 개발한 대용량의 실시간 로그 처리에 특화되어 설계된 오픈 소스 메시지 브로커이다. 아파치 카프카는 원래 링크드인이 개발한 것으로, 2011년 초에 최종적으로 오픈소스화 되었다. Kafka 메시지를 게시하는 Producer, 토픽을 구독하여 데이터를 받는 Consumer로 구성되어 있다. 토픽은 실생활의 주소와 유사하다. 토픽을 약속하고 Producer가 메시지를 보내면 그에 맞는 Topic을 구독하고 있는 Consumer가 메시지를 받는다. Topic은 여러 Broker에 분산되어 저장되며 이렇게 분산된 Topic을 Partition이라고 한다. 어떤 메시지가 Partition에 저장될지는 key를 통해 정해지며, 같은 키를 가지는 이벤트는 항상 같은 Partition에 저장된다. Kafka는 Topic의 partition에 지정된 Consumer가 항상 정확히 동일한 순서로 Partition의 메시지를 읽을 것을 보장한다[12].

3.4 API Gateway

3.4.1 Netflix Zuul

Zuul은 Netflix OSS(Netflix Open Source Software)에 포함된 컴포넌트들 중 하나로써 API 게이트웨이를 구현하는 역할을 한다. Netflix Zuul은 2013년부터 문제없이 운영하고 있으며 구체적인 사례로는 라이엇 게임즈가 운영한 결과를 내보인 적이 있다[13].

3.4.1 Netflix Eureka

Zuul은 Netflix OSS(Netflix Open Source Software)에 포함된 컴포넌트들 중 하나로써 디스커버리 패턴을 구현하는 역할을 한다. Eureka 서버가 서비스 레지스트리를 담당하며 본 논문은 Client-Side Discovery Pattern으로 플랫폼을 구성할 것이다.

IV. 구현

이 장에서는 개념 증명을 구현하기 위한 접근 방식을 자세히 설명한다.

4.1 Platform Architecture

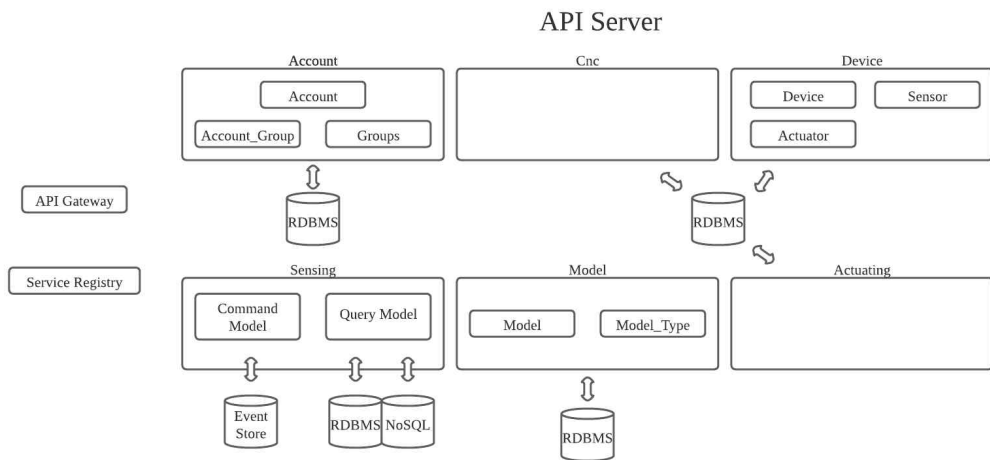


그림 15 IoT Platform Architecture

제안하는 IoT API Server는 그림 15와 같다. 기존의 플랫폼과 크게 달라진 점은 Device 애그리거트, Sensor 애그리거트, Actuator 애그리거트이다. 기존은 Sensor 애그리거트 안에 Sensor 서비스와 Sensing 서비스 또한 Actuator 애그리거트에 Actuator 서비스 Actuating 서비스로 구성되어 있었지만 서비스의 운영에 따라 비즈니스 로직이 변경 및 추가되면서 Device 애그리거트의 Sensing 서비스 및 Actuator 서비스의 잦은 호출로 인한 네트워크 부하가 심해지게 된다. Sensing 서비스와 Actuator 서비스를 Device 애그리거트 안에 포함이 되었다. Sensing 애그리거트를 보면 내부에 Command Model과 Query Model이 존재하고 Command Model에는 Event Store와 DB Connection을 하고 Query Model은 RDBMS와 NoSQL에 대해 DB Connection을 한다. 그림 16은 Sensing 애그리거트의 대한 자세한 그림을 보여준다.

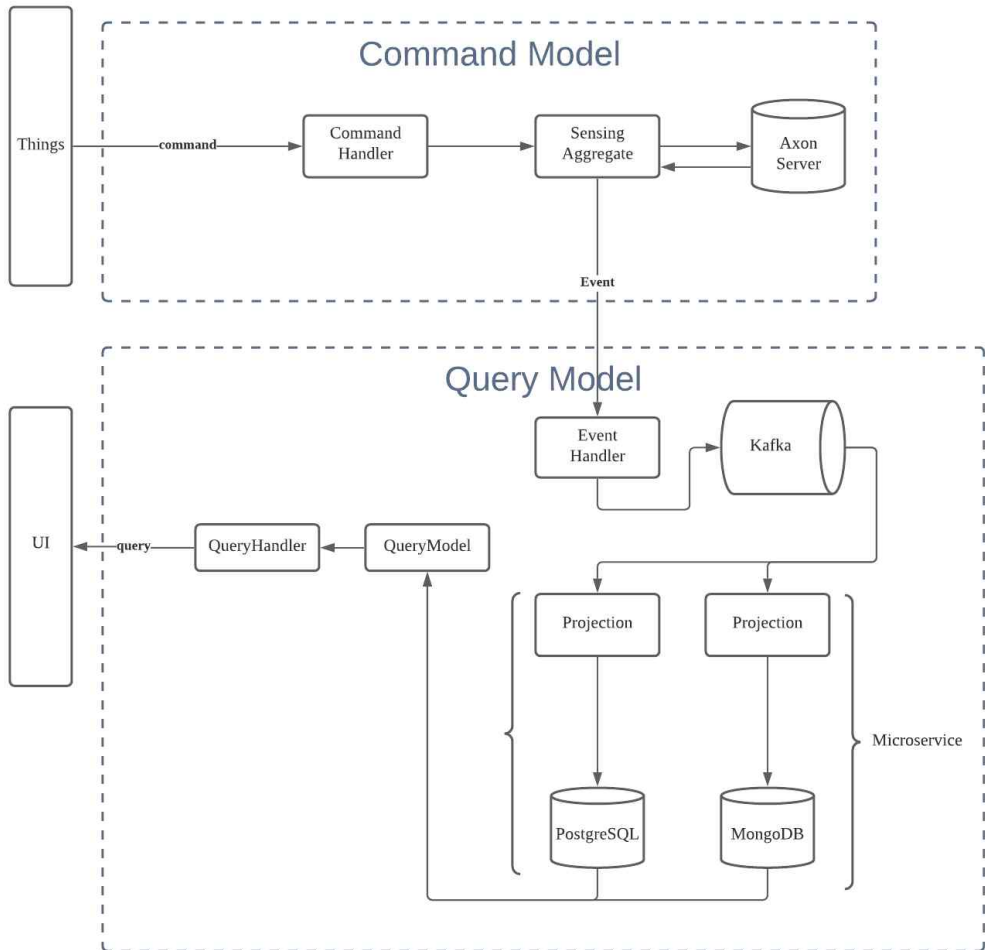


그림 17 Sensing Aggregate CQRS 구조

사물에서 명령을 보내면 Command Handler를 통해 Sensing Aggregate를 거쳐 Axon Server에 이벤트가 저장되며 그와 동시에 Event Handler를 통해 Kafka 메시지 큐에 이벤트를 보낸다. Projection은 Microservice로 되어 있으며 구독된 토픽을 통해 큐에서 이벤트를 전달받아 연결된 데이터 저장소에 저장하게 된다. 자세한 내용은 4.2절과 4.3절을 통해 설명한다.

4.2 Command-Side Architecture

Sensing 애그리거트의 읽기 모델은 그림 17과 같은 형태로 구현되어 있다.

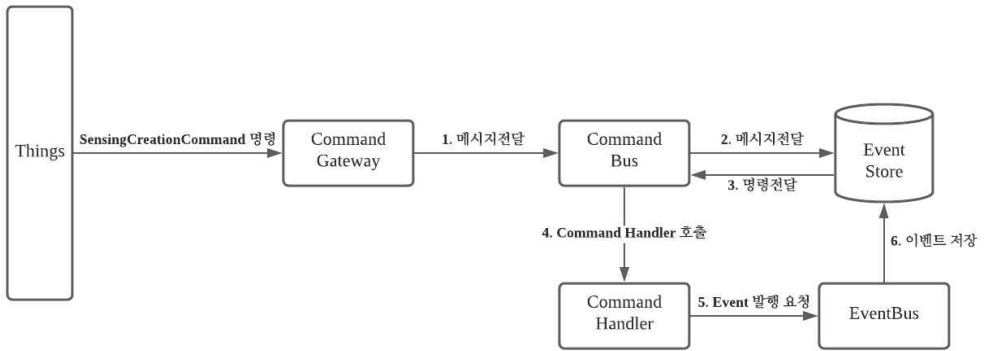


그림 17 Command-Side Architecture

위 그림의 흐름은 다음과 같다. 먼저 센서에서 SensingCreationCommand 명령을 읽기 모델에 보낸다. 명령은 Command Gateway를 거쳐 CommandBus에 메시지를 전달한다. CommandBus는 EventStore에 메시지를 전달하고 EventStore는 CommandBus에 명령을 전달한다. CommandBus는 Command Handler를 호출하고 CommandHandler는 EventBus에게 Event 발행을 요청하고 EventBus는 EventStore에 이벤트를 저장한다.

4.2.1 Command, Event

센서가 보낸 SensingCreationCommand 명령은 다음과 같은 속성으로 되어 있다. SensingId, sensorId, sensingValue, environmentValue로 구성되어 있으며 sensingId와 sensorId는 UUID로 각각 센싱과 센서에 대해 유일하게 식별할 수 있다. sensingTime은 LocalDateTime으로 지정되어 있으며 sensingValue와 environmentValue는 센서에 대한 측정값을 JSON으로 받기위해 HashMap으로 구성된다. 그림 18는 SensingCreationCommand에 대한 구현 코드를 나타낸다.

```
public class SensingCreationCommand {

    private UUID sensingId;
    private UUID sensorId;
    private LocalDateTime sensingTime;
    private HashMap<String, String> sensingValue;
    private HashMap<String, String> environmentValue;
}
```

그림 18 SensingCreationCommand

1. sensingId: 센싱 된 값의 식별할 수 있는 유일한 ID이다.
2. sensorId: 센서를 식별할 수 있는 유일한 ID이다.
3. sensingTime: 센서가 측정한 시간을 나타낸다.
4. sensingValue: 센서가 측정한 데이터의 값이다.
5. environmentValue: 센서의 값이 측정되었을 때 주위에서 영향을 주는 환경 데이터 값이다.

SensingCreationCommand는 Sensing 애그리거트를 통해 Event로 변환된다. 그림 19는 Event에 대한 구현 코드를 나타낸다.

```
public class SensingCreationEvent {

    private UUID sensingId;
    private UUID sensorId;
    private LocalDateTime sensingTime;
    private HashMap<String, String> sensingValue;
    private HashMap<String, String> environmentValue;
}
```

그림 19 SensingCreationEvent

4.2.2 Command DTO

그림 20을 보면 sensingValue의 SugarContent는 당도 센서에서 측정된 값이라고 가정을 했다. 아래의 enviromentValue의 값들을 보면 온도, 습도, 일조량의 값들이 존재한다. 당도에 영향을 받는 값들을 JSON 형식으로 Command Handler에 전달한다.

```
{
  "sensingId": "7bd490f9-e0f9-4a84-a9ac-230429ecb9a1",
  "sensorId": "7bd490f9-e0f9-4a84-a9ac-230429ecb9a2",
  "sensingValue": {
    "sugarContent": "7.0"
  },
  "environmentValue": {
    "temperature" : "30.0",
    "humidity" : "50.1",
    "Sunshine" : "30.0",
    ....
  }
}
```

그림 20 SensingCreationCommand DTO

EventStore의 필드 구성은 그림 21을 통해 설명한다. token은 이벤트가 추가되었다는 것을 식별하는 토큰이며 eventIdentifier는 이벤트를 식별하는 Id aggregateIdentifier는 Aggregate를 식별하는 UUID이다. aggregateSequeneNumber는 Aggregate Sequene Number이며 aggregateType은 Aggregate에 대한 Type을 보여줌으로써 어떠한 Aggregate에 요청을 보냈는지에 대한 이벤트인지 알 수 있다. payloadType은 payloadData는 센서가 보낸 이벤트 값을 나타내며 추후에 읽기 모델에 보내지는 데이터이다.

4.2.3 Event Store Metadata

token	2
eventIdIdentifier	f276c9c3-d669-4acc-8a4c-c0d3af00314f
aggregateIdentifier	49317ed1-6974-4a1d-a914-112ecff67a06
aggregateSequenceNumber	0
aggregateType	SensingCreationAggregate
payloadType	sensing.SensingCreationEvent
payloadRevision	
payloadData	<sensing.SensingCreationEvent><sensingId>49317ed1-6974-4a1d-a914-112ecff67a06</sensingId><sensorId>7bd490f9-e0f9-4a84-a9ac-230429ecb9a1</sensorId><sensingTime>2021-05-01T05:25:02.795609</sensingTime><sensingValue><entry><string>dustValue</string><string>45.0</string></entry></sensingValue><environmentValue><entry><string>temperature</string><string>30.0</string></entry></environmentValue></sensing.SensingCreationEvent>
timestamp	2021-04-30T20:25:02.942Z
metaData	{traceId=6b101950-85e6-4086-a973-99b6d84c113a, correlationId=6b101950-85e6-4086-a973-99b6d84c113a}

그림 21 EventStore Metadata

1. token: 이벤트가 추가되었다는 것을 식별하는 토큰
2. eventIdIdentifier : 이벤트를 식별하는 Id
3. aggregateIdentifier : Aggregate를 식별하는 Id
4. aggregateSequeneNumber : Aggregate 시퀀스 번호
5. aggregateType : Aggregate Type
6. payloadType : 보낸 Event 이름
7. payloadData : Event에 대한 값

4.2 Query-Side Architecture

Sensing 애그리거트의 쓰기 모델은 그림 22와 같은 형태로 구현되어 있다.

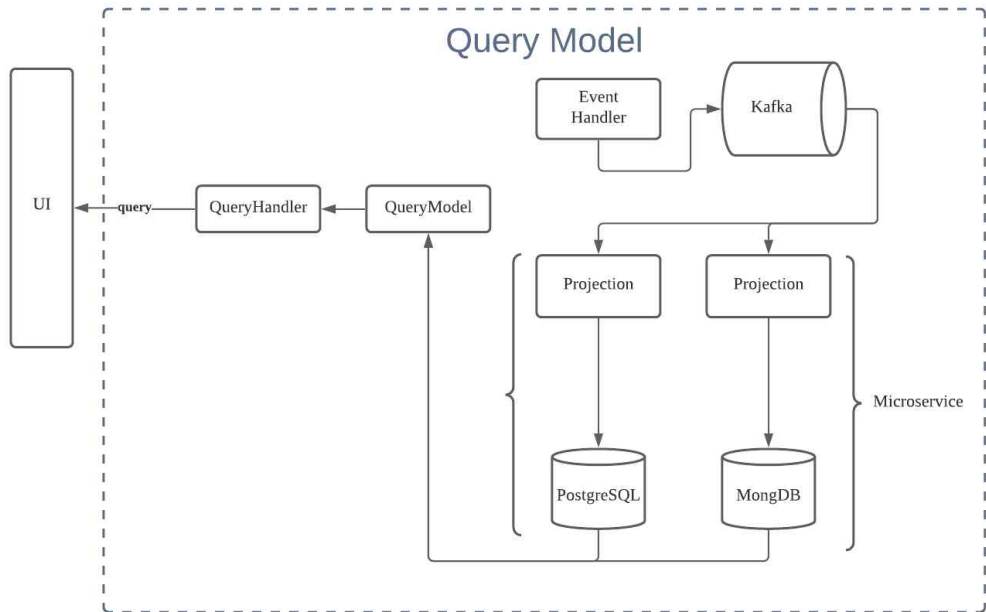


그림 23 QuerySide Architecture

앞서 EventStore에 업데이트된 이벤트가 있으면 일관성을 유지하기 위해 Projection에 연결된 데이터 저장소에 투영된다고 설명하였다. 네트워크 부하를 줄이기 위해 비동기적으로 이벤트를 전송하는 Kafka를 사용하였다. 이벤트를 보내기 위해 EventHandler는 Producer가 되며 각각의 Projection은 Consumer가 되어 이벤트를 전송한다. PostgreSQL이 연결되어 있는 Projection의 토픽 이름을 sensings/sensing-creation/pg라 명명하였고 MongoDB가 연결되어 있는 Projection의 토픽 이름을 sensings/sensing-creation/mongo으로 명명하였다. Projection은 MSA의 구조로 변경하였기 때문에 데이터 저장소를 쉽게 확장하여 구현할 수 있다. QueryModel은 사용자가 요청한 query에 대해 모델을 만들며 QueryHanlder를 통해 사용자에게 보인다.

4.2.1 PostgreSQL Projection



sensing	
sensing_id	uuid
environment_value	jsonb
sensing_time	timestamp
sensing_value	jsonb
sensor_id	uuid

그림 23 Sensing Table

PostgreSQL 데이터 저장소에 만들어진 테이블은 그림 23과 같이 단순히 sensing Table밖에 존재하지 않는다. sensingCreateEvent에 대한 데이터 값만 저장하기 때문인데 필요에 따라 다른 애그리거트 테이블을 추가하여 사용자가 원하고자 하는 쿼리에 따라 조인하여 보여줄 수 있다. Sensing Table에 데이터가 들어가기 위한 흐름은 그림 24의 시퀀스 다이어그램과 같다. 그림을 보면 먼저 API Gateway, SensingEventSourcing, PostgresqlProjection 서비스가 실행되었을 때 Service Registry에 각각의 네트워크가 등록되며 등록되었다는 응답메시지를 보내준다. 마이크로서비스들이 등록이 잘 되었다면 사물은 센싱 값들을 보낼 수가 있다. 플랫폼에 등록된 사물들은 엣지서버를 통하여 API 서버로 통신하게 되지만 Sensing 애그리거트만 설명하기 위해 그 과정을 생략한다. 사물은 정해진 API를 통해서 API Gateway에 SensingCreationCommand 명령을 보낸다. API Gateway는 서비스의 네트워크 위치정보를 Service Registry를 통해 정보를 알 수 있어 SensingCreateCommand를 SensingAggregate에 보낸다. 명령에 대한 응답은 동기 적으로 처리되고 SensingAggregate는 명령을 이벤트로 만들어 sensings/sensing-creation/pg 토픽을 통해 Kafka Queue에 이벤트를 넣는다. Queue에 이벤트가 들어가면 sensings/sensing-creation/pg를 구독한 Consumer가 이벤트를 받을 수가 있다.

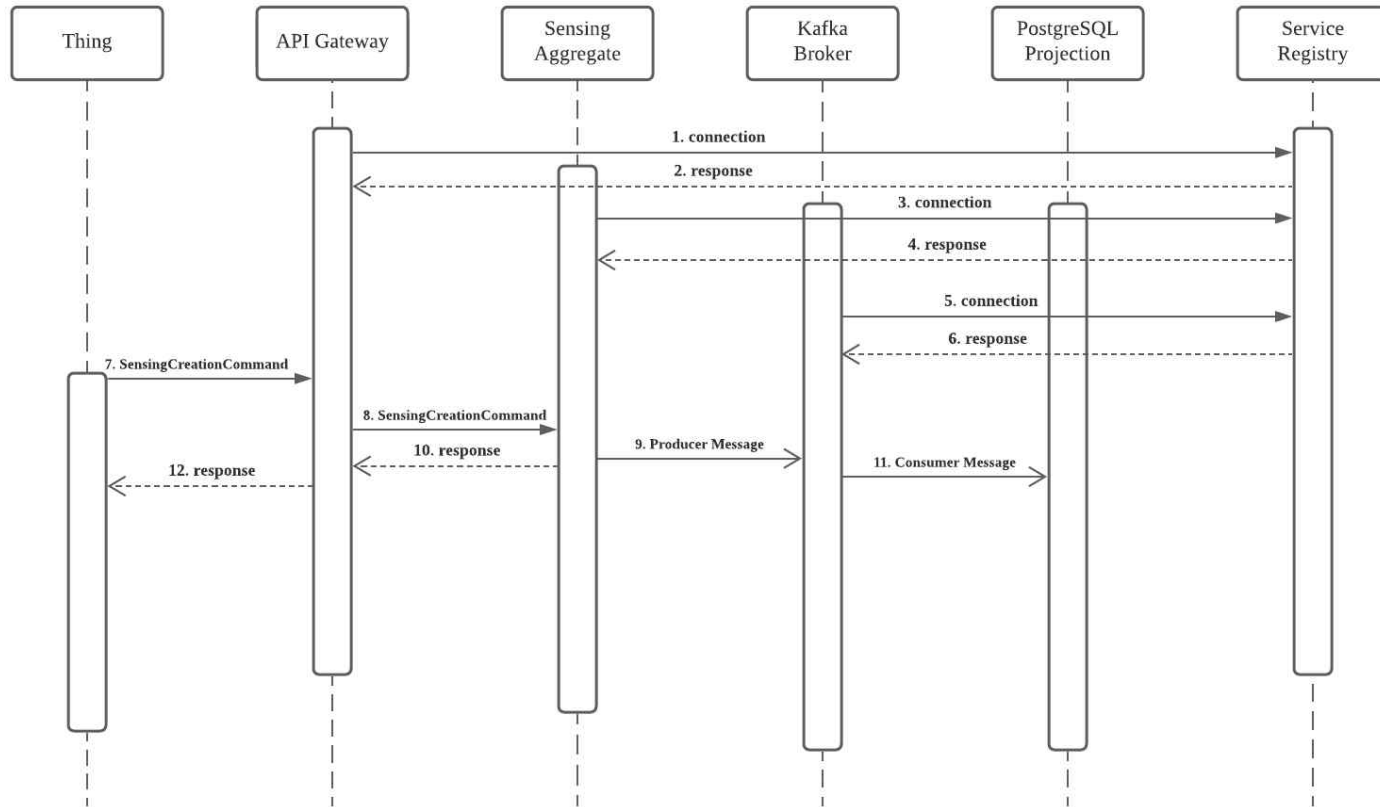


그림 24 PostgreSQL Projection Create에 대한 시퀀스 다이어그램

4.2.2 mongoDB Projection

```
{
  "_id" : BinData(3,"008TH7fpFpEnONx4CrCdpA=="),
  "sensorId" : BinData(3,"hEr54PmQ1HuhuewpBC0sqQ=="),
  "sensingTime" : ISODate("2021-04-08T16:49:51.543Z"),
  "sensingValue" : { "sugarContent": "7.0" },
  "environmentValue" : { "temperature" : "30.0", "humidity" : "50.1", "Sunshine" : "30.0"},
  "_class" : "io.wisoft.eventsourcing.nosql.query.domain.SensingDoc"
}
```

그림 25 Sensing Collection Data

mongoDB는 Table이 따로 없으며 저장되는 데이터의 형식은 그림 25와 같다. _id와 sensorId는 데이터베이스 특성상 UUID가 BinData의 형식으로 저장된다. sensingTime은 ISODate 타입의 UTC로 저장되었으며 sensingValue의 값은 측정된 센싱의 값 environmentValue는 측정된 세션의 값과 관련된 환경 데이터이며 현재 온도, 습도, 일조량이 존재한다. data는 시계열로 저장되며 축적된 데이터는 사용자의 요청의 따라 CSV로 변환하여 제공한다. 그림 26은 분석할 수 있는 형태의 CSV를 보여준다.

	A	B	C
1	sensingValue	environmentValue	sensingTime
2	("sugarContent": "7.0")	("temperature": "30.0", "humidity": "50.1", "Sunshine": "30.0")	2021-04-09T01:49:51.543
3	("sugarContent": "7.1")	("temperature": "30.1", "humidity": "50.1", "Sunshine": "30.0")	2021-04-06T20:37:17.334
4	("sugarContent": "7.1")	("temperature": "30.2", "humidity": "50.1", "Sunshine": "30.0")	2021-04-07T19:07:51.570
5	("sugarContent": "7.0")	("temperature": "30.1", "humidity": "50.1", "Sunshine": "30.0")	2021-04-07T19:07:55.666
6	("sugarContent": "7.0")	("temperature": "30.1", "humidity": "50.1", "Sunshine": "30.0")	2021-04-07T20:49:11.025
7	("sugarContent": "7.0")	("temperature": "30.1", "humidity": "50.1", "Sunshine": "30.0")	2021-04-07T20:49:44.456
8	("sugarContent": "7.0")	("temperature": "30.0", "humidity": "50.1", "Sunshine": "30.0")	2021-04-07T20:51:04.948
9	("sugarContent": "7.0")	("temperature": "30.0", "humidity": "50.1", "Sunshine": "30.0")	2021-04-07T20:51:05.415
10	("sugarContent": "7.0")	("temperature": "30.0", "humidity": "50.1", "Sunshine": "30.0")	2021-04-07T20:51:07.474
11	("sugarContent": "7.0")	("temperature": "30.0", "humidity": "50.1", "Sunshine": "30.0")	2021-04-07T20:51:09.032

그림 26 Sensing Data CSV

그림 28은 MongoDB Projection에 Sensing 데이터가 들어가기 위한 시퀀스 다이어그램이다. 초기화 및 등록 과정은 그림 27과 같으며 kafka Queue에 대한 토픽은 sensings/sensing-creation/mongo로 구독하여 이벤트를 전송한다.

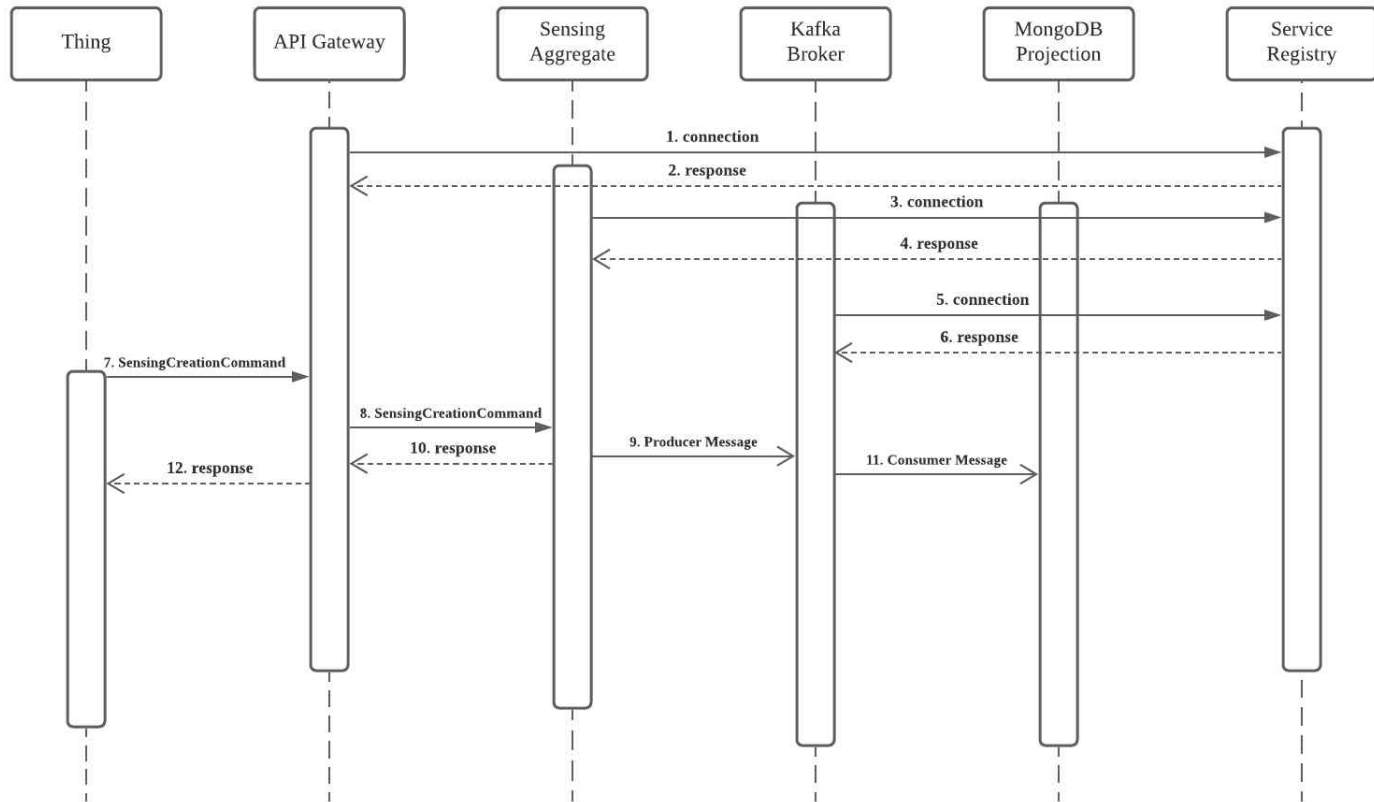


그림 27 MongoDB Projection Create에 대한 시퀀스 다이어그램

V. 검증 및 평가

5.1 구현 요약

사용자가 센서 데이터를 쓰기 모델에 보내는 형태에 따라서 이벤트 스토어에 이벤트 스트림으로 저장된다. 사용자가 필요한 형태에 따라 언제든지 바꿀 수 있다. 읽기 모델과 일관성을 맞추기 위해 이벤트가 쓰기 모델에 들어올 때마다 읽기 모델의 데이터 저장소에 이벤트가 투영된다. 관계형 데이터베이스에서는 최신 데이터를 사용자에게 보여줄 수 있으며 NoSQL 데이터베이스는 사용자가 필요한 형태에 따라 시계열 데이터를 제공할 수 있는 모델을 만들었다.

5.2 실험 내용 및 결과

실험 내용은 다음과 같다. 데이터 저장소를 폴리글랏 구조로 활용하였을 때 구현의 유효성을 알아보는 실험이다. 데이터 저장소를 한 개만 사용하였을 때의 일관성이 도달하는 시간과 데이터 저장소를 두 개 이상 사용하였을 때 일관성이 도달하는 시간이 같아야 한다. 그렇지 않으면 Projection이 늘어날수록 일관성을 도달하는 데 오래 걸리기 때문에 사용자가 최신 데이터를 가져오는데 어려움이 있을 것이다. 실험은 테스트 총 Thing(사물의 수)는 100명개 늘려서 진행하였으며 총 테스트 사례는 5가지로 테스트하였다. 테스트 시작 시간을 측정하고 각각의 Projection 데이터베이스가 최종적으로 일관성을 도달하기까지의 시간을 측정한다.

5.2.1 PostgreSQL Projection만을 연결하였을 때의 최종 일관성

사용자들이 동시에 요청을 수행하였을 때 PostgreSQL Projection이 최종 일관성을 도달하는데 걸리는 지연율을 실험한다.

그림 28은 실험은 쓰기 모델의 이벤트 저장소에서 읽기 모델의 PostgreSQL 데이터베이스 사이의 일관성을 도달하는데 걸리는 시간을 측정한 표이다. 테스트 서버에서 보낸 데이터가 PostgreSQL 데이터베이스에 전부 저장된 시간을 측정하였으며 총 걸린 시간은 테스트 시작 시간 - PostgreSQL Insert 종료 시간을 계산하였다. 각각 총 Thing에 따라 데이터가 전부 Insert 하는 시간이 평균적으로 점차 늘어나는 것을 확인할 수 있다.

	총 Thing	테스트 시작 시간	PostgreSQL Insert 종료 시간	총 걸린 시간
PostgreSQL	100	3:58:50	3:59:94	0:01:04
	200	4:03:42	4:05:12	0:01:36
	300	4:47:07	4:48:56	0:01:49
	400	4:49:50	4:51:57	0:02:07
	500	4:52:39	4:55:02	0:02:03

그림 28 PostgreSQL Projection 최종 일관성 도달 테스트

5.2.2 MongoDB Projection만을 연결하였을 때의 최종 일관성

사용자들이 동시에 요청을 수행하였을 때 MongoDB Projection이 최종 일관성을 도달하는데 걸리는 지연율을 실험한다.

그림 29는 실험은 쓰기 모델의 이벤트 저장소에서 읽기 모델의 PostgreSQL 데이터베이스 사이의 일관성을 도달하는데 걸리는 시간을 측정한 표이다. 테스트 서버에서 보낸 데이터가 MongoDB 데이터베이스에 전부 저장된 시간을 측정하였으며 총 걸린 시간은 테스트 시작 시간 - MongoDB Insert 종료 시간을 계산하였다. 각각 총 Thing에 따라 데이터가 전부 Insert 하는 시간이 평균적으로 점차 늘어나는 것을 확인할 수 있다.

	총 Thing	테스트 시작 시간	MongoDB Insert 종료 시간	총 걸린 시간
MongoDB	100	4:09:10	4:09:42	0:00:32
	200	3:54:18	3:55:21	0:01:03
	300	3:56:20	3:57:35	0:01:15
	400	4:00:54	4:03:17	0:02:07
	500	4:03:41	4:06:04	0:02:23

그림 29 MongoDB Projection 최종 일관성 도달 테스트

5.2.3 PostgreSQL Projection, MongoDB Projection을 연결하였을 때의 최종 일관성

사용자들이 동시에 요청을 수행하였을 때 PostgreSQL Projection과 MongoDB Projection이 최종 일관성을 도달하는데 걸리는 지연율을 실험한다.

그림 30은 PostgreSQL Projection과 MongoDB Projection을 동시에 연결하여 일관성을 도달한 결과를 나타낸 표이다. 테스트 시작 시간을 기록하여 각각 Projection 별로 데이터가 최종적으로 Insert된 시간을 기록하였으며 그에 따라 총 걸린 시간을 측정하였다.

	총 Thing	테스트 시작 시간	PostgreSQL Insert 종료 시간	MongDB Insert 종료 시간	PostgreSQL 총 걸린 시간	MongoDB 총 걸린 시간
PostgreSQL,MongoDB	100	4:13:53	4:14:57	4:14:25	0:01:04	0:00:32
	200	4:20:58	4:22:34	4:22:02	0:01:36	0:01:04
	300	4:25:37	4:27:17	4:26:53	0:01:40	0:01:16
	400	4:28:39	4:30:46	4:30:02	0:02:07	0:01:23
	500	4:33:38	4:35:42	4:35:14	0:02:04	0:01:36

그림 30 동시적으로 요청 시 최종 일관성 도달 테스트

5.2.4 실험 결과

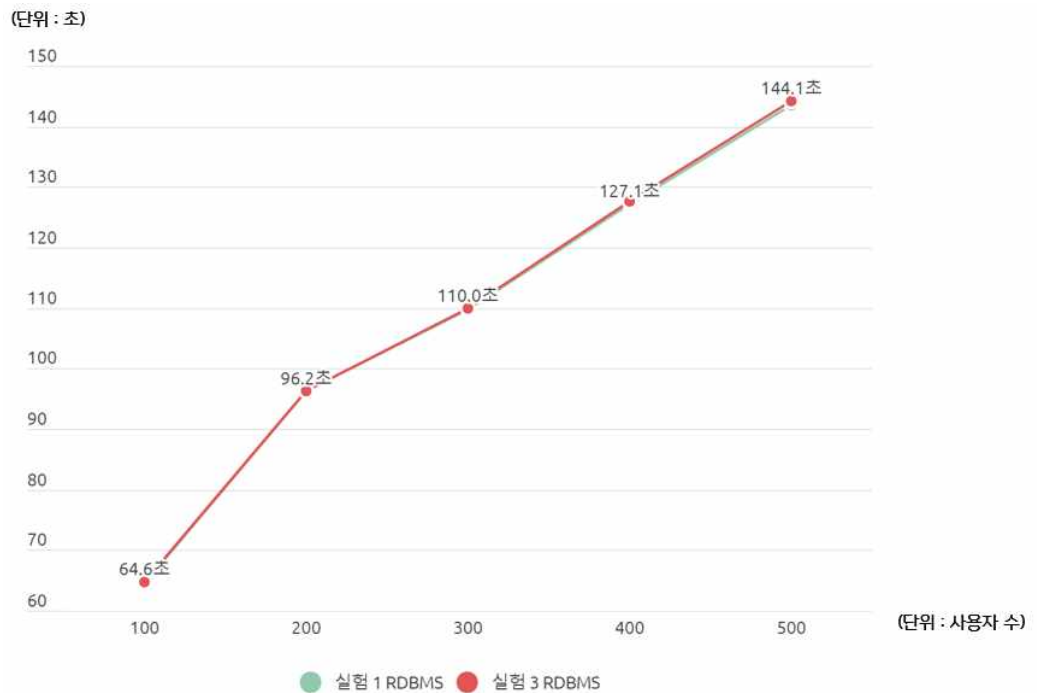


그림 31 실험 1과 실험 3의 RDBMS 비교

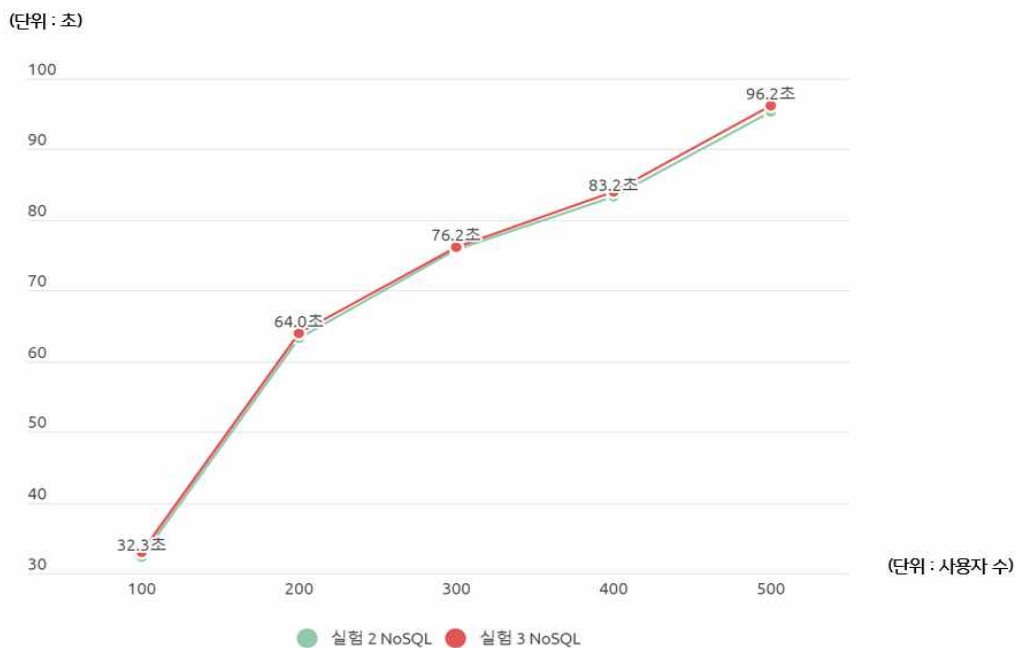


그림 32 실험 2와 실험 3의 NoSQL 비교

먼저 그림 31을 봤을 때 테스트 PC의 작업량에 따라 일관성이 도달되는 시간의 차이는 미세하게 존재하지만 실험 1 PostgreSQL Projection과 실험 3의 PostgreSQL Projection이 일관성에 도달하는데 걸리는 시간이 거의 같다고 볼 수 있다. 그림 32도 마찬가지로 실험 2 MongoDB Projection과 실험 3의 MongoDB Projection이 Thing에 따라 일관성에 도달하는데 걸리는 시간이 늘어나지만 Projection을 동시에 여러 개를 접근한다 해도 전체적으로 데이터베이스 성능의 영향에는 지장이 없는 모습을 보인다.

VI. 결론 및 향후 연구

1. 결론

본 연구의 목적은 CQRS 패턴과, EventSourcing을 활용하여 데이터 저장소를 폴리글랏 구조로 활용하였을 경우 대용량 데이터의 활용과 폴리글랏 구조의 선택이 시스템 구현에 어떠한 영향을 미칠 수 있는지 조사하는 것이었다. 데이터를 필요에 따라 사용자에게 효율적으로 제공하는 개념 증명 작업을 수행하였다. 또한 폴리글랏 구조를 사용함으로써 같은 이벤트 형식을 보내더라도 마이크로서비스로 분리되어 있기 때문에 데이터를 투영시키는 것에 시스템 성능에 영향이 없다는 것을 확인했다. 이로 인해 데이터 저장소가 추가되더라도 전체 시스템의 성능에는 크게 영향이 없는 것으로 보인다.

2. 향후 연구

일관성을 도달하는 시간을 개선하기 위한 추가 연구가 수행될 수 있다. 폴리글랏 구조를 활용함에 앞서 사용자가 이벤트를 보내 여러 데이터베이스에 저장을 하여도 성능의 문제는 없다는 것을 확인하였지만 계속적으로 이벤트 스트림을 보내면 사용자가 최종 데이터를 읽는 시간이 오래 걸릴 것이다. 또한 일관성을 도달함에 있어 데이터가 유실의 여부도 있기 때문에 성능과 데이터의 안정성 연구도 추가적으로 진행되어야 한다.

V. 참 고 문 헌

- [1] 2019~2025년 전 세계 IoT와 Non-IoT 연결 수 전망, IoT Analytics 2021
- [2] Martin Fowler, “CQRS Pattern,” <https://martinfowler.com/bliki/CQRS.html>, [Accessed online, April 13 2021].
- [3] G. Young, “Cqrs documents by greg young,” Young, vol. 56, 2010.
- [4] Martin Fowler, “Aggregate,” https://martinfowler.com/bliki/DDD_Aggregate.html, [Accessed online, April 14 2021].
- [45] Gilbert, Seth, and Nancy Lynch. “Perspectives on the CAP Theorem.” Computer 45.2 (2012): 30-36.
- [6] Chris Richardson of Eventuate, “Service Discovery,” <https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture>, [Accessed online, April 16 2021].
- [7] 전철호, 서보민, 이근혁, 전현식, 박현주. (2018). MSA가 적용된 서비스에서 생산되는 데이터의 효율적인 분석을 위한 연구. 한국정보과학회 학술발표논문집, (), 1144-1146.
- [8] AxonFramework, <https://axoniq.io/product-overview/axon-framework#0>, [Accessed online, April 19 2021].
- [9] AxonServer, <https://axoniq.io/product-overview/axon-server>, [Accessed online, April 19 2021].
- [10] PostgreSQL, <https://www.postgresql.org/>, [Accessed online, April 20 2021].
- [11] MongoDB, <https://www.mongodb.com/what-is-mongodb/>, [Accessed online, April 22 2021].
- [12] Apache Kafka, <https://kafka.apache.org/>, [Accessed online, April 23 2021].

[13] Riot Games, <https://technology.riotgames.com/news/riot-games-api-fulfilling-zuuls-destiny/>, [Accessed online, April 24 2021].

ABSTRACT

Polyglot structure design to provide users with the efficiency of analysis within IoT systems with EventSourcing and CQRS

Min Young Park

Dept. of Mobile Convergence Engineering

Graduate School of Information & Communication,

HANBAT National University

Advisor : Hyun-Ju Park

As IoT technology progresses, the amount of data continues to grow as the number of devices being connected increases. After all, the amount of load is increasing as the service continues. In addition, since most systems are often composed of structured architectures, it is difficult to obtain quality analysis results even if a large amount of data is collected. To reduce load balancing and efficiently provide large amounts of data, in this paper, we demonstrate an architecture suitable for large data environments. Changing to an architecture using EventSourcing, a pattern that stores both CQRS patterns and event logs from devices, the database divides services into MSAs to create polyglot structures, using RDBMS, a relational database, and MongoDB, for analysis. The proposed platform aims to utilize large amounts of data in various environments qualitatively and efficiently for data analysis.