공학석사학위논문

실시간 교통빅데이터 시뮬레이션을 위한 IoT 시뮬레이터 설계 및 구현

Design and Implementation of IoT Simulator for Real-Time Traffic Bigdata Simulation

충북대학교 대학원

빅데이터협동과정 빅데이터전공

안 상 희

2017년 2월

공학석사학위논문

실시간 교통빅데이터 시뮬레이션을 위한 IoT 시뮬레이터 설계 및 구현

Design and Implementation of IoT Simulator for Real-Time Traffic Bigdata Simulation

지도교수 조 완 섭

빅데이터협동과정 빅데이터전공

안 상 희

이 논문을 공학석사학위 논문으로 제출함

2017년 2월

본 논문을 안상희의 공학석사학위 논문으로 인정함.

심사위원장 최 상 현 ㊞

심 사 위 원 조 완 섭 ㊞

심 사 위 원 이 경 희 ㊞

충 북 대 학 교 대 학 원

2017년 2월

차 례

Design and Implementation of IoT Simulator for Real-Time Traffic Bigdata Simulation

Sang Hee An

Department of BigData

Graduate School, Chungbuk National University

Cheongju, Korea

Supervised by Professor Cho, Wan-Sup

Abstract

As the various types of devices that collect IoT-based traffic stream data increase, the amount of real-time data is increasing every year. Due to these characteristics, it is difficult to save, manage and analyze with existing traffic management systems. Therefore, big data technology should be employed in the traffic management systems. However, there is plenty of risks in the transition into the big data system because of the mission-critical traffic management systems in the sites. For this reason, it is necessary to implement IoT simulator based on the actual traffic data to test the stability of the big data platform, real-time analysis without

interfering with the existing traffic management systems.

In this research, based on the IoT protocol MQTT, we implemented a web based IoT simulator and designed to be able to simulate using actual traffic data in Cheongju city. The execution time of the simulator and the message transfer amount per second has been measured, and the original performance evaluation was carried out. As a result of the performance evaluation, the IoT simulator generates and transfers 120 messages per second and it can cover 10 million of DSRC messages per day, which seems to be sufficient capability for Cheongju city. Moreover, we implemented a visual interface based on Marey graph for real-time bus operation monitoring, and several patterns of bus operation have been found in the city.

Keyword: Traffic Big Data, Simulator, IoT, MQTT, Visualization

표 차례

그림 차례

Ⅰ. 서 론

1.1 연구배경 및 목적

최근 디지털 환경에서 사용자의 참여 기회가 증가하고, 보다 쉬운 접속성을 가진 IT 기술의 발전으로 인해, 데이터의 양이 급속히 증가하고 있다. IDC 디지털 유니버스(IDC Digital Universe)의 보고서에 따르면 2011년에 생산되고 복제된 정보의 양이 1.8제타바이트(Zettabytes)에 달하고 있으며, 매 2년마다 그 양이 2배씩 증가한다고 발표하였다. 이러한 엄청난 양의 데이터의 축적은 비단 웹에서뿐만 아니라 일반적인 기업에서도 동일하게 발생하고 있다. 맥킨지(Mckinsey)의 보고서에 따르면 2009년을 기준으로 미국의 많은 산업영역에서 종업원 1,000명 이상을 보유한 기업들의 평균 데이터 보유량이 1페타바이트(Petabytes)를 넘는 것으로 조사되었다. 이와 더불어 수백만개의 네트워크 구조로 연결되어 정보를 생산해 내는 센서 영역과 게놈 지도와 같은 의료 및 공공 영역 등 다양한 분야에서 매일 방대한 양의 데이터가 축적되고 있다.

이러한 데이터의 폭증은 기존의 분석 및 처리를 위한 IT 기술과 데이터베이스의 역량을 초과하는 상황이다. 이처럼 기존의 분석 및 지원 기술로는 감당하기 어려울 만큼 축적되고 생산되는 데이터의 홍수를 빅데이터(Big Data)라 일컫는다. 빅데이터에 대한 정의가 무엇인가에 대한 정확한 합의가 되지는 않았지만, 빅데이터에 대해 1테라바이트 이상의 데이터로 정의를 하는 의견이 있기도 하다. 하지만 단순히 빅데이터에 대한 정의를 데이터의 양으로 규정짓기는 어렵다. 왜냐하면 산업에 따라 처리하는 데이터의 양에 대한 기준이 서

로 다를 뿐 아니라, 데이터의 양 또한 기술이 발전함에 따라 빅데이터라 불리는 데이터 집합의 크기도 점차 늘어나기 때문이다.

이러한 빅데이터의 활용분야는 다양하며, 각 분야별로 고유한 특성을 가지고 있다. 교통 분야를 포함하여 의료, 과학, 공공, 제조, 금융 등에서 빅데이터가 활용되고 있거나 활용될 전망이다 (정덕원, 2014).

교통 분야에서는 ICT기술의 발전에 따라, 과거 지점 위주의 데이터 수집, 표본 추출에 의한 모수 추정, 중앙집중형 데이터 관리 및 제공 시스템에서 빅데이터를 접목하여 발전하는 방향으로 변화하고 있다. 빅데이터 수집 범위도 점(point)에서 면(area)으로의 확산되고, 추정이 필요 없는 모집단에 근접한 자료 수집 가능, 비정형화된 데이터 수집 및 제공 가능, 개별 맞춤형 정보 제공이 가능할 정도로 다양하고 큰 데이터가 수집, 유통, 활용되고 있다. 즉, 최근 들어 교통 분야도 급속하게 빅데이터 시대로 전환되고 있다 (이석주 외, 2013).

교통 분야에서는 각종 IoT기반 센서로부터 다양한 데이터가 실시간으로 발생한다. 이러한 데이터는 끊이지 않고 발생하므로 일반적으로 스트림 데이터(Stream Data)라 부른다. 스트림 데이터는 쉼 없이 빠르게 생성되며(Velocity), 하루하루 쌓이게 되면 대용량의 데이터가 되고(Volume), 다양한 디바이스로부터 다양한 형식으로 발생한다(Variety). 이러한 스트림 센서 데이터는 빅데이터의 특성을 가지므로 기존 교통 운영시스템에서 저장, 관리, 분석하기에는 어려움이 있다. 이에 따라 빅데이터 환경의 교통 운영시스템으로 전환할 필요가 있다. 하지만, 빅데이터 플랫폼의 안정성과 실시간 수용력 등이 입증되지 않은 상태에서, 실제 교통 운영시스템에 빅데이터 기술을 적용하는

것은 안정성을 해칠 수 있어 신중하게 접근해야 한다. 이러한 한계점을 극복하고자 기존 교통 운영 시스템에 지장을 주지 않고, 축적된 실제 데이터를 기반으로 빅데이터 플랫폼의 실시간 수용력과 실시간 분석 및 시각화 등의 테스트를 하기 위한 IoT 시뮬레이터 구현이 필요한 것이다.

본 연구에서는 IoT 프로토콜인 MQTT를 바탕으로 웹 기반 IoT 시뮬레이터를 구현하고, 청주시 실제 교통 데이터를 사용하여 시뮬레이션을 할 수 있도록 하였다. 그리고 시뮬레이터의 실행 시간, 초당 메시지 전송량을 측정하여 성능 평가를 진행하고, 이 시뮬레이터를 사용하여 스트리밍 데이터의 분석결과를 시각화하는 부분을 소개한다.

본 논문의 구성은 다음과 같다. 제 2장에서는 교통 빅데이터 수집 시스템과 IoT 프로토콜인 MQTT 및 Apache Kafka에 대하여 설명한다. 제 3장에서는 IoT 시뮬레이터 설계 및 구현에 대하여 설명한다. 제 4장에서는 시뮬레이터의 실행 시간, 초당 메시지 전송량을 측정하여 성능 평가를 진행한다. 제 5장은 실제로 시뮬레이터에서 생성한 스트리밍 데이터가 빅데이터 플랫폼을 거쳐 분석 및 시각화하는 부분을 소개한다. 그리고 제 6장에서는 결론을 맺는다.

Ⅱ. 이론적 배경 및 관련연구

본 장에서는 시뮬레이터 개발과 관련된 내용을 소개한다. 먼저 2.1절에서는 교통 빅데이터 수집 시스템과 교통 데이터의 종류에 대해 설명한다. 그 다음으로 2.2절과 2.3절에서 시뮬레이터의 구성요소인 IoT 프로토콜인 MQTT와 Apache Kafka에 대해 설명한다.

2.1 교통 빅데이터 수집 시스템

청주시에서는 2004년 이후 교통카드시스템, 버스정보시스템(BIS), 첨단교통관리시스템(ATMS) 등으로부터 다양한 교통데이터가 생성되고 있다. 최근들어 빅데이터 기술을 활용하여 이를 수집하여 저장하고 통합분석 할 수 있는 분석 플랫폼을 구축하였으며(2015), 분석결과는 과학적 교통정책 수립하는데 활용되고 있다(김상호, 2015).

청주시 교통빅데이터 시스템에 축적되고 있는 각 데이터에 대하여 간략히 설명하면 다음과 같다. 교통카드 데이터는 승객이 승·하차할 때 찍는 카드의 거래 내역 데이터이고, 버스운행기록(BIS) 데이터는 버스운행 시 교차로 혹은 정류장의 진·출입 시점의 버스운행기록과 관련된 데이터이다. 그리고 첨단교통관리(ATMS) 데이터는 도시부 간선도로를 중심으로 데이터를 수집하고 있으며, DSRC와 AVI로 구분하여 수집하고 있다.

[표 1]은 청주시에서 수집되고 있는 교통 데이터의 현황을 보여주고 있다.

[표 1] 교통데이터 현황

	교통카드	버스운행기록(BIS)	첨단교통관리(ATMS)	
			DSRC	AVI
데이터생성 시점	승객하차 시 교통카드 단말기 입력 시점	버스운행 시 각 정류장 또는 교차로 진 출입 시점	OBU 단말기 부착 차량 수집 교차로 통과 시	전 차량 수집 교차로 통과 시
데이터 수집기	청주시 등록된 공영시내버스	2112개(정류장과 일부 교차로)	103곳	6곳
수집주기	2016년 이후 실시간 수집	실시간 수집	실시간 수집	실시간 수집
수집범위	청주시 공영시내 버스 승·하차, 환승 거래 데이터	청주시 공영시내 버스 운행기록	통합청주시 권역	청주시 일부 도로 구간
데이터발생 건수/일	약 23만/일	약 35만/일	약 230만/일	약 90만/일
데이터수집 위치	(주)마이비사 서버	청주시 BIS 서버	청주시 DSRC 서버	청주시 ATMS 서버
데이터 관리	(주)마이비사	청주시	청주시	청주시

청주시의 첨단교통관리시스템(ATMS)은 도로, 차량 등 교통체계의 구성요소에 전자 정보 통신 제어 등 첨단기술을 융·복합하고 실시간 교통정보를 개발, 활용하여 이용자에게 실시간 교통정보를 제공하고 관리자에게 효율적인 교통관리 및 정책수립을 지원하도록 하는 서비스이다.

ATMS의 개념도는 [그림 1]과 같으며, 크게 3가지로 구성되는데 교통정보수집시스템, 교통정보가공시스템, 교통정보제공시스템이다. 교통수집시스템은 도로소통 및 교통량 등을 파악하기 위하여 노변 장치(DSRC) 103개소, 영상검

지기(VDS) 3개소, 차량번호인식장치(AVI) 6개소, 동영상수집장치(CCTV), 교차로감시카메라 2개소를 통해 교통정보를 수집하고 있다. 이 수집된 데이터를 교통가공시스템을 통하여 버스정보시스템과 융합하거나 데이터베이스의 저장한다. 이후 교통정보제공시스템을 통하여 도로전광표지(VMS)와 홈페이지나 모바일을 통해 정보를 제공한다.

[그림 1] 청주시 첨단교통관리시스템 개념도

출처:http://www.cjits.go.kr/w/introduce/service.html

2.2 MQTT 프로토콜

MQTT(Message Queue Telemetry Transport)는 1999년 IBM에서 개발된 프로토콜로써 원격 측정이나 스마트폰에서와 같이 제한된 컴퓨팅 성능과 네트워크 연결 환경에서의 동작을 고려하여 설계된 대용량 메시지 전달 프로토콜

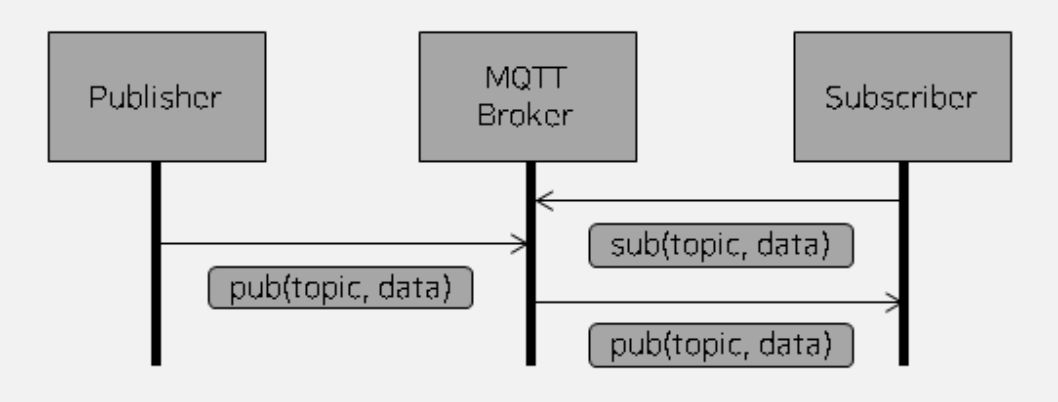
이다. 2013년엔 OASIS(Organization for the Advancement of Structured Information Standards)에 의해서 사물 인터넷을 위한 메시지 프로토콜로 선정되었을 뿐만 아니라 배터리 에너지 소모에 측면에서 효율적인 프로토콜임이 검증되고 있다(M. Prihodko, 2012; The MQTT protocol).

또한 개방적이고 비교적 쉽게 구현할 수 있으며 간단하고 경량의 Publish/subscribe 구조로 설계되었기 때문에 천 개 이상의 클라이언트가 하나의 서버에서 지원된다. 이러한 특성들로 인해 MQTT 프로토콜은 낮은 대역폭이나 높은 지연시간을 갖는 네트워크 및 제한된 처리능력과 메모리를 갖는 장치들로 구성된 환경에서 사용되기에 적합하다.

2.2.1 MQTT 프로토콜 특징

(1) Publish/Subscribe

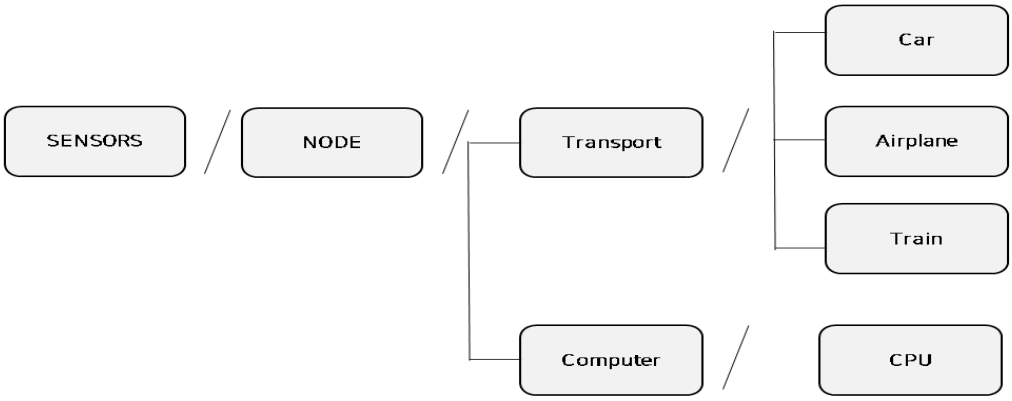
MQTT 프로토콜의 가장 큰 특징은 Publish/Subscribe 구조를 가진다는 점이다. 일반적으로 특정 Topic에 대해 메시지를 발행(Publish)하는 발행자(Publisher)와 관심 있는 메시지를 구독(Subscribe)하는 구독자(Subscriber)와 프록시 서버와 같이 중간에서 중개역할을 하는 브로커(Broker)로 구성되어 있다. 아래 [그림 2]는 Publisher와 Subscriber가 통신하는 기본 구조를 나타낸 것이다. 하나 이상의 Subscriber가 특정 Topic을 Subscribe한 상태라면 Publisher에 의해 특정 Topic으로 Publish된 데이터는 MQTT Broker를 거쳐 Subscriber에게 전달된다.



[그림 2] 기본적인 Publish/Subscribe 구조

(2) Topic Format

MQTT에서 특정한 토픽으로 발행된 메시지들은 같은 대상을 가진 하나의 영역으로 간주되어 클라이언트는 토픽에 가입함으로써 해당 토픽에 대한 메시지를 수신한다. 기본적으로 MQTT에서 토픽은 계층적인 구조를 가지며, 토픽의 구성에서 계층을 나누는 기준은 문자 (“/”)를 이용한다. 아래 [그림 3]은 계층적 토픽 구성의 예이다. 예를 들어 Car와 관련된 토픽을 구독하고 싶을 경우 ‘SENSORS/NODE/Trasport/Car’ 토픽을 브로커에 요청하여 구독하면 된다.

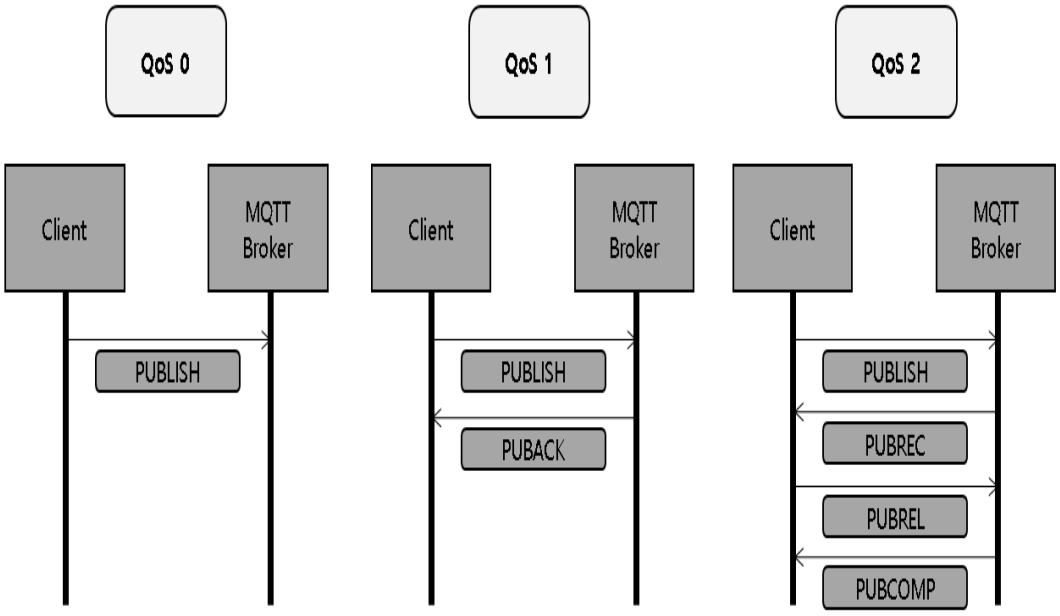


[그림 3] 계층적 토픽 구성 예시

(3) QoS(Quality of Service)

MQTT는 메시지 처리에 대한 신뢰성을 보장하기 위해서 3가지 레벨의 QoS를 지원한다. 높은 수준의 QoS는 보다 안정적인 메시지 전달을 보장하지만 이로 인해 더 많은 네트워크 대역폭을 사용하거나 메시지의 지연시간을 늘어난다.

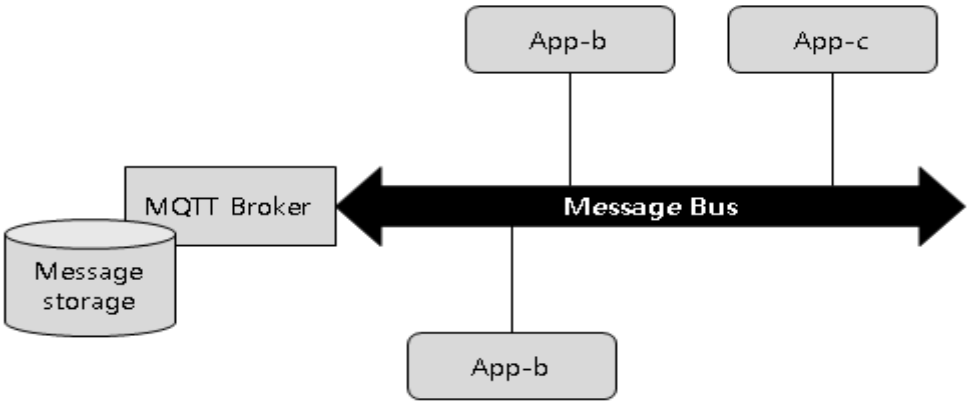
QoS 레벨의 특징은 살펴보면, QoS 0은 메시지를 한 번만 전달하고 전달 여부는 확인하지 않는다. 따라서 큰 페이로드를 가지는 메시지일 경우, 패킷 손실이 발생하면 메시지가 전달되지 않고 유실될 가능성이 높다. QoS 1은 메시지를 최소 1번이상 전달하고 전달여부를 확인한다. 하지만 PUB ACK 패킷이 유실 될 경우 메시지가 불필요하게 중복으로 전달될 가능성이 있다. QoS 2는 4-way handshake를 통해 정확하게 한 번만 전달되지만 종단 간 지연이 늘어나는 단점이 있다. 아래 [그림 4]는 QoS 레벨에 따른 패킷 교환 방법이다.



[그림 4] QoS 레벨에 따른 패킷 교환 방법

(4) 메시지 버스(Message Bus)

MQTT는 메시지 버스 시스템이다. MQTT Broker가 메시지 버스를 만들고 메시지를 보내면, 버스에 붙은 구독자(혹은 Application)들이 메시지를 읽어가는 방식이다. 아래 [그림 5]는 메시지 버스 흐름도이다.

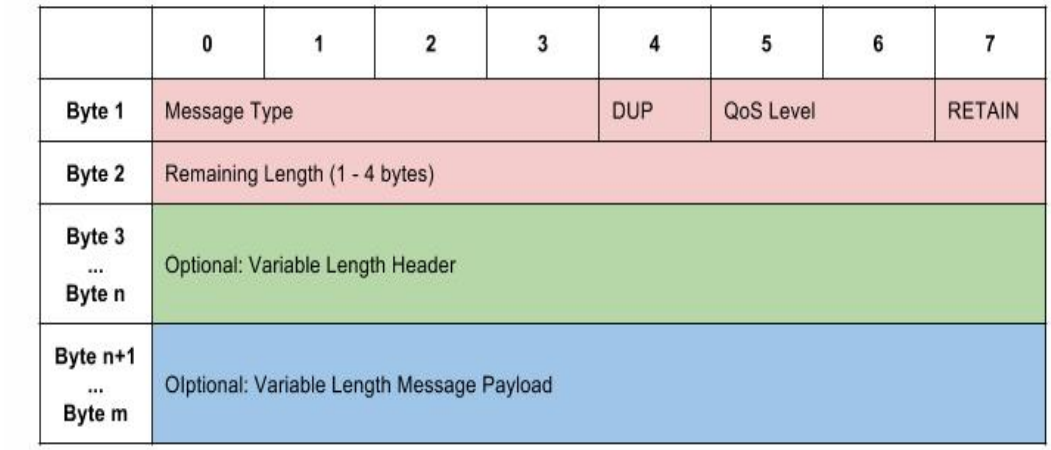


[그림 5] 메시지 버스 흐름도

2.2.2 MQTT 프로토콜 구조

MQTT 프로토콜은 [그림 6]처럼 크게 3가지 부분으로 이루어져 있다. 첫 번째는 Fixed Header(고정 헤더) 부분으로 8개의 비트 2묶음, 총 2개의 바이트로 구성되며 구성 요소는 Message Type, 플래그들(DUP, QoS Level, RETAIN)과 메시지 크기(Remaining Length)로 구성된다. 다음으로 Variable Header(가변 헤더)는 Fixed header와 Payload 사이에 존재하며 각 기능에 따라 조금씩 달라진다. 일반적으로 payload의 메시지 크기 정보를 담고 있다(MQTT Tutorial).

마지막으로 Payload 부분은 실제 데이터(메시지)를 담고 있는 영역이다. 하나 이상의 UTF-8로 인코딩 된 문자열을 포함하고 있으며 이 문자열은 클라이언트의 토픽 및 사용자 이름과 암호 등의 식별자를 지정한다.



	0	1	2	3	4	5	6	7
Byte 1	Message Type				DUP	QoS Level		RETAIN
Byte 2	Remaining Length (1 - 4 bytes)							
Byte 3 ... Byte n	Optional: Variable Length Header							
Byte n+1 ... Byte m	Olptional: Variable Length Message Payload							

[그림 6] MQTT 프로토콜 구조

2.2.3 MQTT Broker의 종류와 장단점

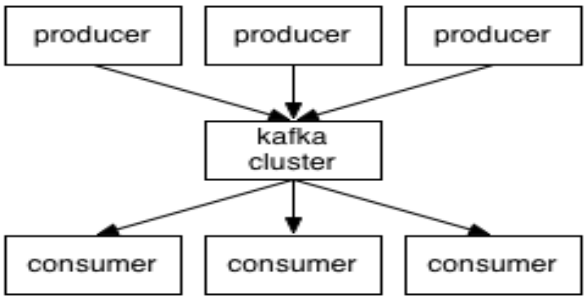
MQTT는 다양한 종류의 Broker를 가지고 있다. 그 중에서도 가장 대중적으로 활용되는 RabbitMQ와 Mosquitto를 비교한다. 먼저 RabbitMQ Broker는 MQTT, HTTP, STOMP, Web Socket 등의 다양한 통신과 클러스터링을 지원하는 장점이 있지만, 프로그램이 무겁기 때문에 소형 디바이스에 사용하기에는 무리가 있다는 단점이 있다. Mosquitto Broker는 MQTT 전용 Broker로 경량이며 Broker가 가져야 할 대부분의 기능을 충실히 지원한다는 장점이 있지만, 단일 프로토콜만 지원하는 단점이 있다. 본 연구에서는 추후 교통 센서 프로토콜의 확장 가능성과, 클러스터링 등을 고려하여 RabbitMQ Broker를 사용한다(김상현 2016). [표 2]는 RabbitMQ와 Mosquitto를 비교한 표이다.

[표 2] RabbitMQ와 Mosquitto 비교

특징	RabbitMQ	Mosquitto
다양한 프로토콜 지원	O	X
클러스터링 지원	O	X
경량의 프로그램	X	O
와일드 카드 지원	O	O
3가지 레벨의 QoS 지원	X(QoS 0,1만 지원)	O

2.3 Apache Kafka

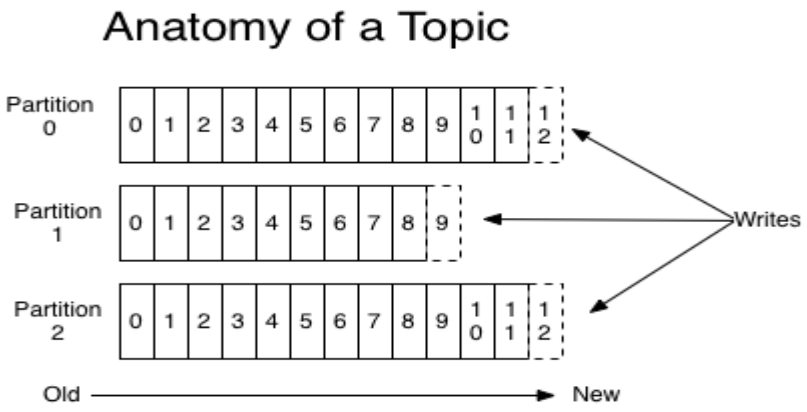
Apache Kafka는 LinkedIn에서 개발된 분산 메시징 시스템으로써 2011년에 오픈소스로 공개되었다. 대용량의 실시간 로그처리에 특화된 분산 Publish Subscribe Messaging System이다. Kafka는 [그림 7]과 같이 크게 Producer, Consumer, Broker로 구성된다.



[그림 7] Kafka 아키텍처

출처: (http://kafka.apache.org/081/documentation.html)

Producer는 메시지를 생성하고 Broker에게 전달하며, broker는 producer에게 전달 받은 메시지를 토픽 별로 관리한다. Consumer는 Broker에 저장 관리되는 메시지를 읽어 처리하는 역할을 한다. 아래 [그림 8]과 같이 Kafka에서 토픽은 파티션 단위로 쪼개져 클러스터의 각 서버들에게 분산하여 저장하고, 각각의 파티션은 1씩 증가하는 오프셋 값을 메시지에 할당하여 각각의 메시지를 파티션 번호와 오프셋 값을 이용하여 식별한다.



[그림 8] Kafka Partition

출처: (http://kafka.apache.org/081/documentation.html)

본 연구에서 Kafka는 MQTT Broker 기준으로 Subscriber역할을 하며, 스트림 데이터를 전달받아 빅데이터 플랫폼인 하둡(Hadoop)과 스팍(Spark)의 분산 처리한다.

Ⅲ. IoT 시뮬레이터 설계 및 구현

본 장에서는 IoT 프로토콜인 MQTT를 기반으로 하여 IoT 시뮬레이터의 설계와 구현에 대해 설명한다. 3.1절에서는 시뮬레이터 시스템 구조에 대해 설명하고, 3.2절에서 실제 데이터를 MQTT Broker에 Publish/Subscribe 하기 위한 MQTT Topic 설계에 대해 설명한다. 마지막으로 3.3절에서 웹 어플리케이션의 설계·구현에 대해 설명한다.

3.1 시스템 구조 및 설계

IoT 센서 시뮬레이터 시스템의 구성요소는 사용자(Client), MS-SQL 서버, 웹 서버(Web Server), MQTT Broker 서버, Kafka Broker 서버로 구성된다. 구성요소와 주요기능은 [표 3]과 같다.

[표 3] 시스템 구성 요소

구성요소	주요기능
사용자(Client)	웹 서버를 접속하여, 시뮬레이터를 사용하는 객체
MS-SQL 서버	교통 데이터를 저장하고 있는 객체
웹 서버(Web Server)	사용자 인터페이스를 제공하고 브로커 서버와 통신하는 객체
MQTT Broker 서버	다양한 프로토콜에 대응하여 연결된 개체 사이의 메시지를 중개 하는 객체
Kafka Broker 서버	메시지 분산 저장하는 객체

(1) 사용자(Client)는 웹 서버에 접속하여 시뮬레이터 정책에 따라 시뮬레이션 기능을 설정하고, 각종 서버의 기본적인 기능 등을 조작하고 제어하는 객체이다.

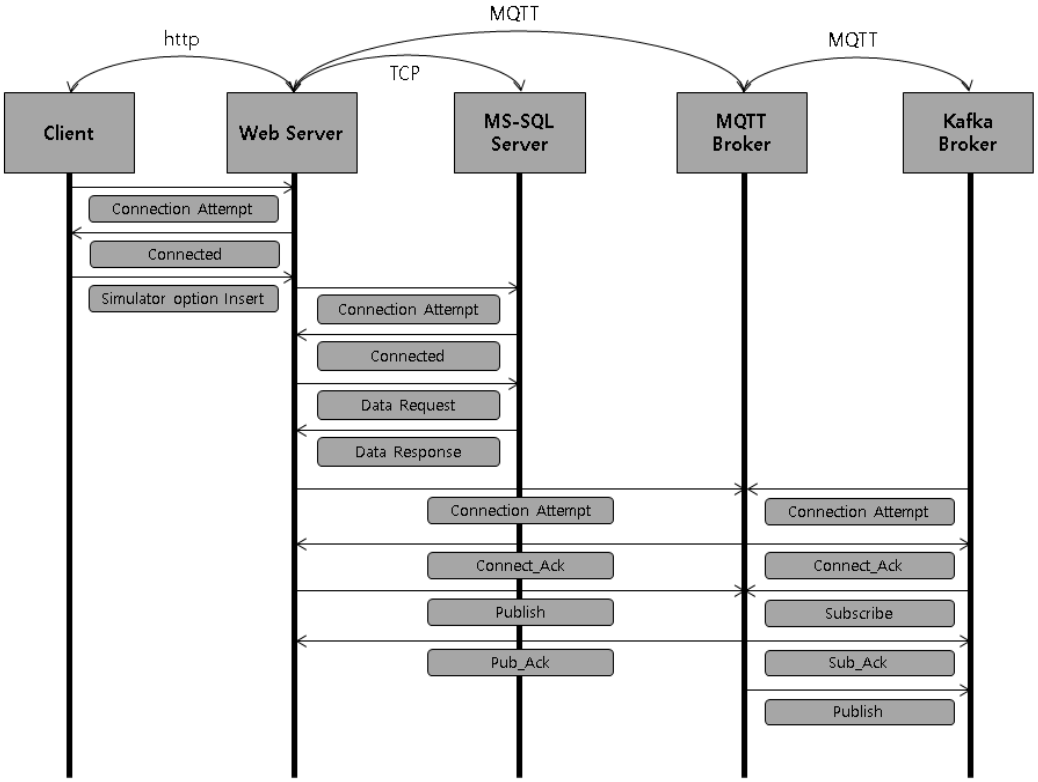
(2) MS-SQL 서버는 교통 데이터베이스에 수집된 데이터를 저장하고 있고, 웹 서버의 요청에 따라 데이터베이스 커넥션을 통하여 데이터를 전달하는 객체이다.

(3) 웹 서버는 사용자에게 시뮬레이터 옵션 선택을 위한 인터페이스를 제공하고, 입력 받은 옵션을 토대로 MS-SQL 서버에 데이터를 요청하며, 수신된 데이터들을 MQTT 브로커(Broker) 서버에 데이터를 토픽(Topic)에 따라 Publish 하는 기능을 한다.

(4) MQTT 브로커(Broker) 서버는 연결된 Publisher와 Subscriber의 중개인으로써 Exchange-Queue 형태로 메시지를 관리하며, 메시지 버스를 이용해 사물과 사물간 통신을 지원한다. 메시지 버스에는 “Topic”이라는 채널을 만들어 일대일, 일대다, 다대다의 메시지 전송을 가능하게 해준다.

(5) Kafka 브로커(Broker) 서버는 MQTT 브로커에 Subscribe 하는 객체이며, 구독한 메시지를 Kafka 클러스터를 통하여 분산 저장하는 기능을 한다.

위에서 언급한 구성요소에 따라 시스템의 전반적인 프로세스를 크게 놓고 보면 [그림 9]와 같다.

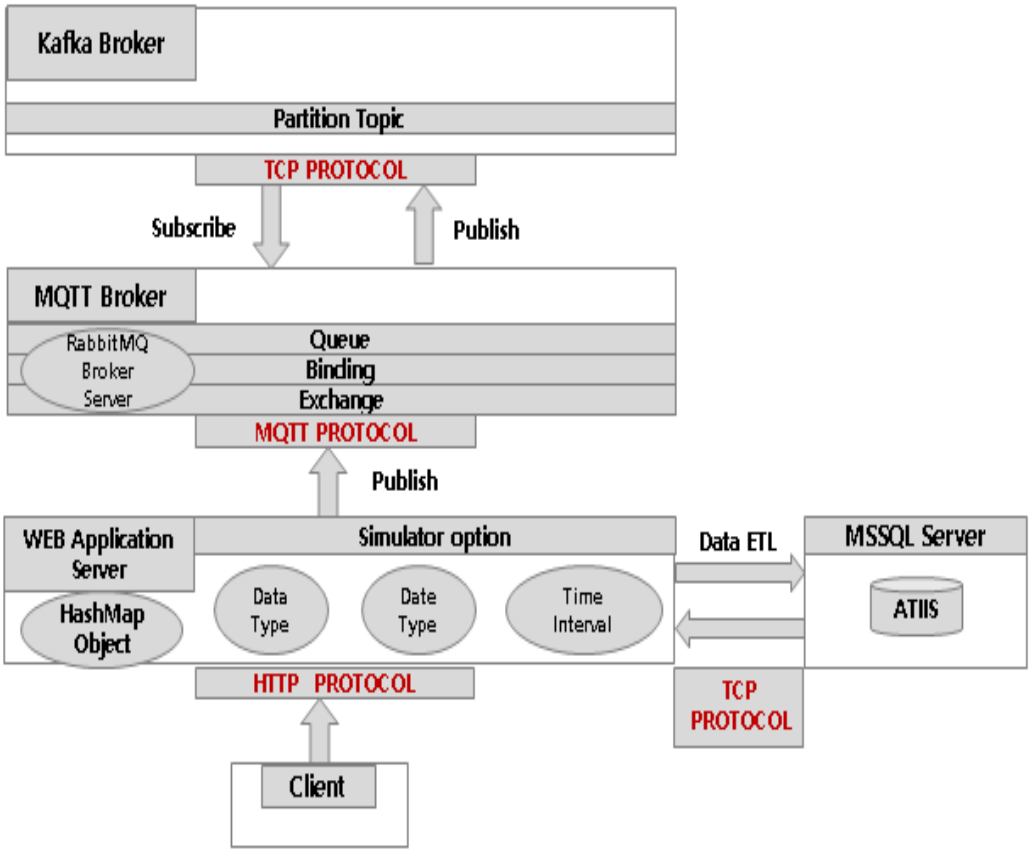


[그림 9] 시스템 프로세스 흐름

사용자(Client)는 웹 서버에 http를 통하여 연결 요청을 한다. 웹 서버에 연결이 된 사용자(Client)는 시뮬레이터 옵션을 선택하여, Start 버튼을 누르면 JDBC는 MS-SQL 서버에 연결을 요청을 한다. MS-SQL 서버의 연결이 된 웹 서버는 MS-SQL 서버의 SQL 쿼리를 전달하게 되고, MS-SQL 서버는 반환된 데이터들을 웹 서버에 전달하게 된다. 반환된 데이터들은 데이터 타입별로 웹 서버에서 HashMap 구조로 임시 저장 한다. 그 이후 웹 서버는 MQTT Broker에 연결 요청을 하고, 연결이 되면 웹 서버는 데이터의 종류에 따라 Topic으로 구분하여, MQTT Broker에 Publish를 한다. 이 때 Kafka Broker도

관심 있는 Topic을 Subscribe한다. 이 후 새로운 데이터가 들어 올 때마다 MQTT Broker는 메시지 버스를 통하여 Kafka Broker의 메시지를 전달한다.

위의 내용을 종합해 시스템의 전체적인 개념도를 그려보면 [그림 10]과 같다.



[그림 10] IoT 시뮬레이터 시스템 개념도

3.2 MQTT Broker 설계

본 논문에서는 사용하는 RabbitMQ Broker는 ISO(ISO/IEC 19464) 표준 AMQP(Advanced Message Queueing Protocol) 메시지 브로커 소프트웨어(Message Broker Software)로 Open Source로 공개되어 있다. 또한, RabbitMQ는 Erlang언어와 Java언어로 만들어졌을 뿐만 아니라, Clustering과 Failover를 위한 OTP 프레임워크로 서버가 구현되어 있다. RabbitMQ는 대용량 데이터를 처리하기 위한 배치작업이나 채팅서비스, 비동기 데이터를 처리의 장점이 있는 Broker이다. 본 연구에서는 RabbitMQ Broker를 설치하고 환경 설정을 중심으로 기술한다.

먼저 RabbitMQ 홈페이지에서 Window Server용 설치 파일을 다운 받는다. Broker를 설치하기 전에 얼랭(Erlang)이 기존에 설치되어야 한다. 얼랭(Erlang) 홈페이지에서 Broker 서버 운영환경에 맞는 버전을 설치한다. 이후 RabbitMQ에서 다운받은 파일을 통하여 설치한다. RabbitMQ에서 제공하는 Management Plugin을 활성화하기 위하여 커맨드 창에 [그림 11]과 같이 입력한다.

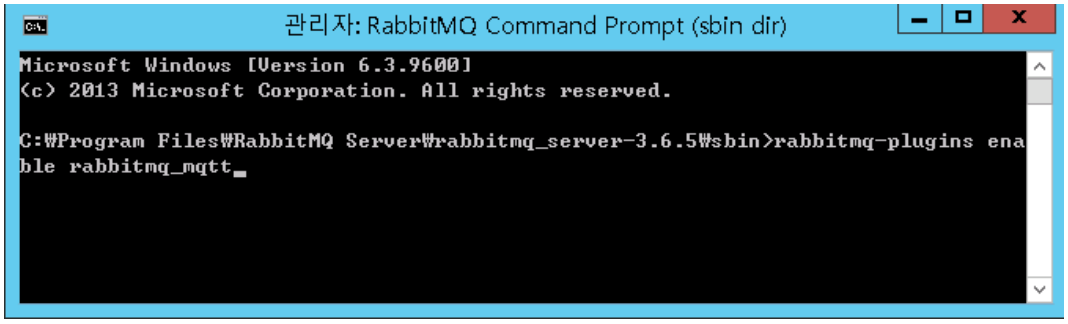
[그림 11] RabbitMQ Management Plugin 활성화 명령어 화면

이후 RabbitMQ Broker에 웹 서버와 Kafka가 접속할 수 있도록 계정을 생성하고 권한을 부여해야 한다. [그림 12]는 계정을 생성하는 명령어 화면이고, [그림 13]은 계정의 권한을 부여하는 명령어 화면이다.

[그림 12] RabbitMQ 계정 생성 명령어 화면

[그림 13] RabbitMQ 권한 부여 명령어 화면

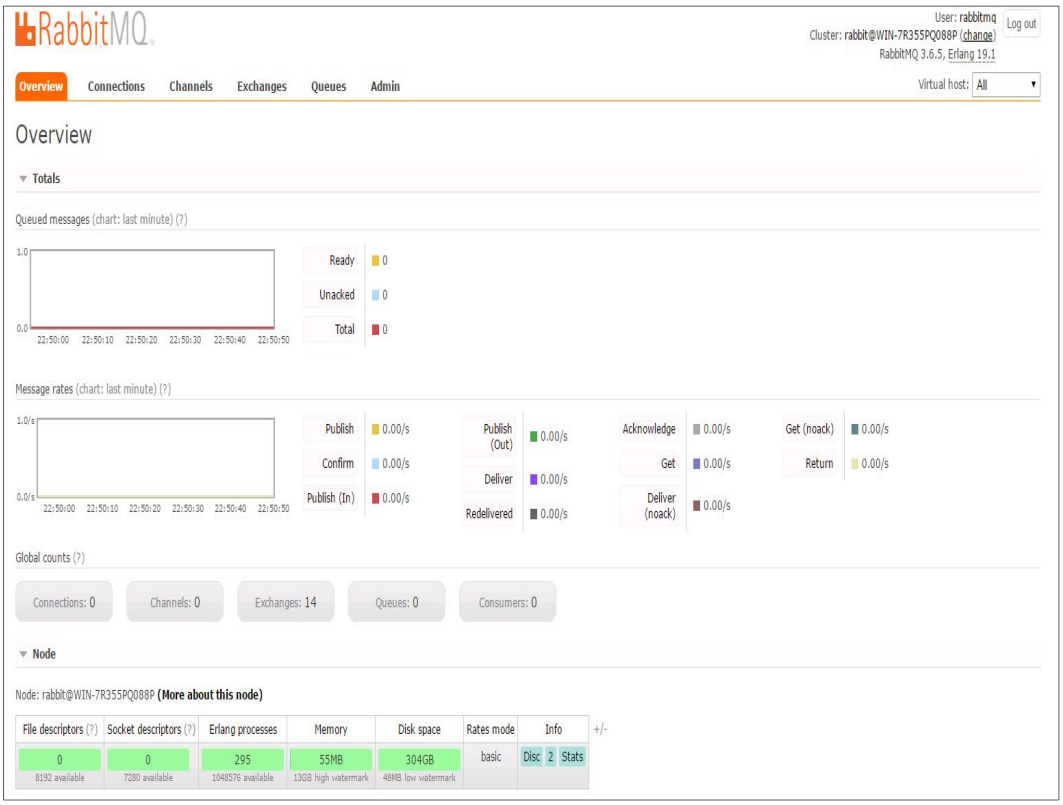
RabbitMQ는 AMQP 프로토콜을 기본적으로 사용하기 때문에 MQTT 프로토콜을 사용하기 위해선 RabbitMQ에서 MQTT 프로토콜을 활성화를 해야 한다. [그림 14]는 MQTT 프로토콜 활성화 명령어 화면이다.



[그림 14] RabbitMQ MQTT 프로토콜 활성화 명령어 화면

Broker 서버가 구동하게 되면 RabbitMQ Plugin(웹 관리자 콘솔)에 Broker 서버의 주소로 접속이 가능하다. 기본 포트는 '15672'이고, 'http://서버주소:15672' 로 접속할 수 있다.

[그림 15]는 RabbitMQ Plugin 접속한 메인 화면이다. RabbitMQ Plugin은 Connection, Channel, Queue, Exchange의 상태와 메시지의 전송현황(Publish), 서버 노드의 기본적인 상태를 모니터링 할 수 있다. 또한, 웹에서 계정 설정 및 가상 호스트 설정, 브로커 서버 정책 설정을 제공하여 일부는 서버의 재시작 없이 브로커의 기본적인 설정을 할 수 있다.



[그림 15] RabbitMQ Plugin에 접속한 메인 화면

3.3 MQTT Topic 설계

MS-SQL에서 가져온 데이터들을 토대로 주제를 분류하여 MQTT Topic을 설계하였다. Publisher는 교통 데이터 항목별로 Topic을 출판(Publish)하고, Subscriber는 관심 있는 교통 데이터 항목을 구독(Subscribe)한다. [표 4]에서 보는 것과 같이 교통 데이터 종류 별 Topic을 간략하게 설계했다. [표 5]는 MQTT 기본 설정에 대한 그림이다.

[표 4] Topic 설계

Topic 설계	
DATA TYPE	TOPIC
CARD	atis-mybi
BIS	atis-bis
DSRC	atis-dsrc
AVI	atis-avi

[표 5] MQTT 기본 설정

MQTT 기본 설정	
Exchange	amq.topic
MQTT_Qos	1
Virtual Host	/

본 연구에서 MQTT QoS는 QoS 1로 설정하였다. 그 이유는 앞서 설명한대로 QoS 0은 메시지를 한 번만 전달하고 전달 여부는 확인하지 않고, 큰 페이로드를 가지는 메시지일 경우 패킷 손실이 발생하면 메시지가 전달되지 않고 유실될 가능성이 높기 때문에 문제가 있다. 즉, 시뮬레이션 특성상 메시지 손실의 가능성이 있기 때문에 신뢰도 측면에서 제외 시켰다.

QoS 2는 4-way handshake를 통해 정확하게 한 번만 전달하지만 종단 간 지연이 늘어나기 때문에, 실시간으로 지속적인 데이터가 들어올 경우 센서(장비)와 MQTT Broker의 과부하가 올 수 있고, 실시간을 추구하는 교통 스트림 데이터의 특성에 맞지 않기 때문에 제외시켰다.

QoS 1은 메시지를 최소 1번 이상 전달하고 전달여부를 확인하고, 만약 PUB ACK 패킷이 유실 될 경우 메시지가 불필요하게 중복으로 전달될 가능성이 있지만, 교통 스트림 데이터의 특징인 실시간성, 신뢰도, 안정성 등의 복합적인 상황을 고려하여 Qos 1을 사용하였다.

3.4 웹 어플리케이션 설계 및 구현

웹 어플리케이션(시뮬레이터)은 기본적으로 스프링 프레임워크(Spring Framework)를 활용하여 설계 및 구현하였다.

웹 어플리케이션을 세부적으로 살펴보면, 웹 인터페이스는 HTML5와 CSS, 부트스트랩(Bootstrap), Thymeleaf를 활용하여 MVC 디자인패턴으로 설계 하였고, Publish/Subscribe의 모듈의 구현은 MQTT, MQTT-SN전용 오픈 소스 라이브러리인 Paho API를 활용하였다. 마지막으로, 웹 어플리케이션은 교통 데이터의 종류에 따라 각각의 스레드(Thread)가 독립적으로 실행되도록 멀티스레드(MultiThread) 방식으로 설계 및 구현하였다.

다음에서 웹 어플리케이션의 중요한 개념인 스프링 프레임워크(Spring Framework)과 MVC 디자인패턴에 대해 간략하게 소개한다.

(1) 스프링 프레임워크(Spring Framework)

스프링 프레임워크는 자바 플랫폼을 위한 오픈소스 애플리케이션 프레임워크로써 간단히 스프링(Spring)이라고도 불린다. 동적인 웹 사이트를 개발하기

위한 여러 가지 서비스를 제공하고 있다. 대한민국 공공기간의 웹 서비스 개발시 사용을 권장하고 있는 전자정부 표준프레임워크의 기반 기술로써 쓰이고 있다. 스프링의 특징으로는 아래와 같다(Introduction to the Spring Framework).

스프링은 경량 컨테이너로써 자바 객체를 직접 관리한다. 각각의 객체 생성, 소멸과 같은 라이프 사이클을 관리하며 스프링으로부터 필요한 객체를 얻어올 수 있다.

스프링은 POJO(Plain Old Java Object) 방식의 프레임워크이다. 일반적인 J2EE 프레임워크에 비해 구현을 위해 특정한 인터페이스를 구현하거나 상속을 받을 필요가 없어 기존에 존재하는 라이브러리 등을 지원하기에 용이하고 객체가 가볍다.

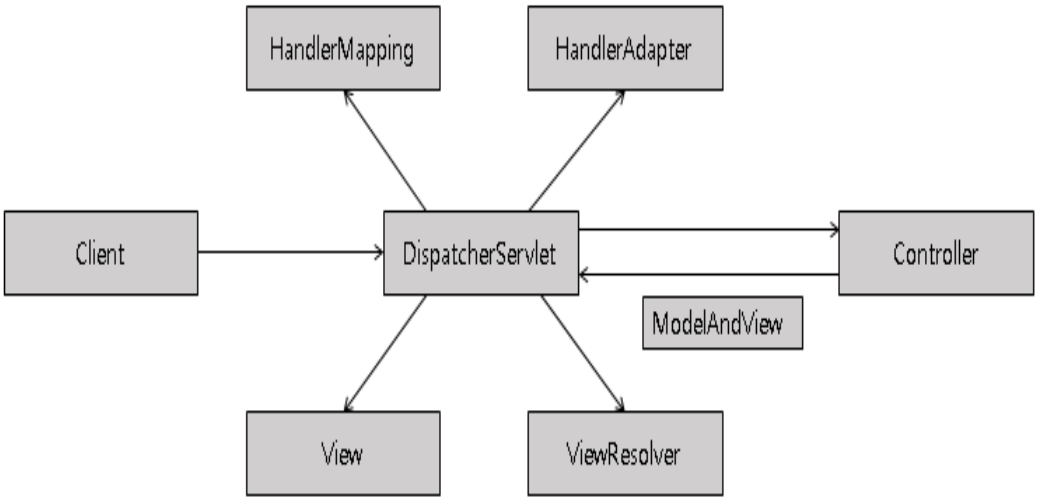
스프링은 제어 반전(IoC: Inversion of Control)을 지원한다. 컨트롤의 제어권이 사용자가 아니라 프레임워크에 있어서 필요에 따라 스프링에서 사용자의 코드를 호출한다.

스프링은 의존성 주입(DI: Dependency Injection)을 지원한다. 각각의 계층이나 서비스들 간에 의존성이 존재할 경우 프레임워크가 서로 연결시켜준다.

스프링은 관점 지향 프로그래밍(AOP: Aspect-Oriented Programming)을 지원한다. 따라서 트랜잭션이나 로깅, 보안과 같이 여러 모듈에서 공통적으로 사용하는 기능의 경우 해당 기능을 분리하여 관리할 수 있다.

(2) MVC 디자인 패턴

MVC 패턴은 Model-View-Controller의 약자로써 소프트웨어 공학에서 사용되는 아키텍처 패턴이다. MVC에서 모델(Model)은 애플리케이션의 정보(데이터)를 나타내며, 뷰(View)는 텍스트, 체크박스 항목 등과 같은 사용자 인터페이스 요소를 나타내고, 컨트롤러(Controller)는 데이터와 비즈니스 로직 사이의 상호동작을 관리한다. [그림 16]은 스프링에서의 MVC 처리 흐름을 나타낸다.



[그림 16] 스프링에서 MVC 처리 흐름도

각 구성요소의 특징을 알아보면 “DispatcherServlet”은 클라이언트의 요청을 전달받아 요청에 맞는 컨트롤러(Controller)가 리턴한 결과 값을 뷰(View)에 전달하여 알맞은 응답을 생성하는 역할을 하고, “HandlerMapping”은 클라이

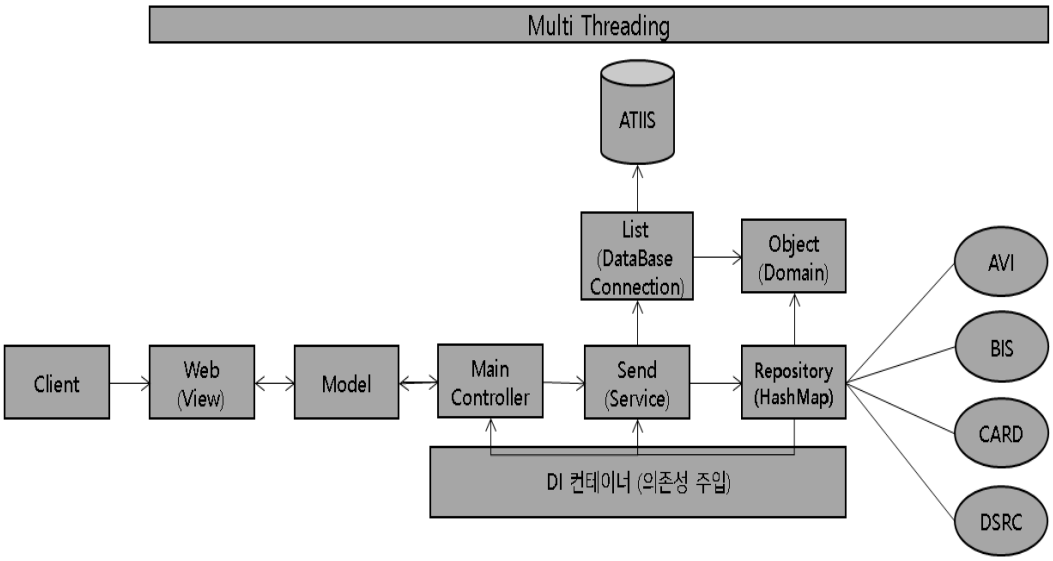
언트의 요청 URL을 어떤 컨트롤러가 처리할지 결정하는 역할을 한다. “HandlerAdapter”는 “DispatcherServlet“이 ServletType의 컨트롤러를 호출하는 역할을 하며, “Controller”는 클라이언트의 요청을 처리한 뒤, 결과를 “DispatcherServlet”에게 리턴 하는 역할을 한다. “ModelAndView”는 컨트롤러가 처리한 결과 정보 및 뷰(View) 선택에 필요한 정보를 담는 역할을 하며, “ViewResolver”는 컨트롤러의 처리 결과를 생성할 뷰(View)를 결정하는 역할을 한다. “View”는 컨트롤러의 처리 결과 화면을 생성하는 역할을 한다.

3.4.1 웹 어플리케이션 프로세스

본 연구에서의 웹 어플리케이션(시뮬레이터)의 프로세스 흐름은 [그림 17]과 같다. 사용자는 XML 파일을 통하여, 데이터베이스 설정, MQTT Broker 설정, Topic 설정 등을 하게 된다. 이후, Web Server에 접속하여, 시뮬레이터 옵션을 선택하게 된다. 시뮬레이터 옵션은 크게 3가지를 설정할 수 있는데, 교통 데이터 타입, 날짜, 시간 간격을 설정한다. 3가지 설정 중 시간 간격에 대해 설명하자면, MS-SQL에서 데이터를 가져온 후, MQTT Broker에 몇 초 간격으로 데이터를 보내는지(Publish)에 대한 설정이다. 예를 들어 4가지의 교통 데이터에 대하여 1초를 선택하면, 초당 약 4건의 데이터를 MQTT 브로커에 Publish 한다. 만약 0초를 입력하면, 교통 데이터 4종류가 시간 순서대로 시뮬레이터의 시간복잡도(N)으로 MQTT Broker에 Publish 한다.

웹 인터페이스에서 옵션으로 설정한 값들은 Model객체의 전달되고, Main Controller가 이 입력 값을 받게 된다. List 클래스는 MS-SQL 서버의 데이터

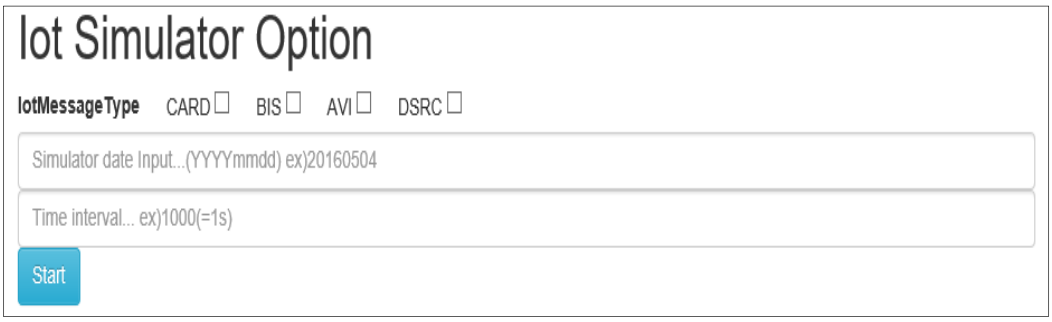
베이스에 연결하여 SQL 쿼리를 전달하고, 쿼리의 해당되는 교통 데이터를 Repository 클래스에서 Domain 별로 HashMap 자료구조 형태로 임시 저장한다. 임시로 저장된 스트림 데이터들은, Send 클래스에서 JSON 포맷으로 변환하여, MQTT Broker에 Topic 별로 앞서 입력된 시간 간격에 따라 Publish한다.



[그림 17] 웹 어플리케이션 프로세스 흐름도

3.4.2 웹 인터페이스 구현

웹 인터페이스는 HTML5, CSS, 부트스트랩(Bootstrap), Thymeleaf를 활용하여 설계하였다. “http:서버주소/index”에 접속하면 [그림 18]과 같이 사용자 인터페이스가 보이게 된다. 교통 데이터 종류와 날짜, 시간 간격을 설정하고 Start를 누르면 시뮬레이터가 작동하게 된다.



[그림 18] 웹 사용자 인터페이스

3.4.3 XML 설정

사용자는 시뮬레이션의 앞서, XML 설정을 해주어야 한다. 크게 설정해야 하는 부분은 2가지로 나뉜다. [그림 19]는 XML 설정 화면이다.

(1) MS-SQL 설정

MS-SQL의 접속 계정 정보와, 서버 주소, 데이터베이스를 설정을 할 수 있다.

(2) MQTT Broker 설정

Broker 접속 계정 정보와 가상호스트, Topic 설정, Exchange 설정을 할 수 있다.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment>MQTT</comment>
    <entry key="MSSQL_ID">ID</entry>
    <entry key="MSSQL_PW">PASSWORD</entry>
    <entry key="JDBC">com.microsoft.sqlserver.jdbc.SQLServerDriver</entry>
    <entry key="MSSQL_URL">jdbc:sqlserver://[redacted]</entry>
    <entry key="MQTT_HOST">[redacted]</entry>
    <entry key="MQTT_PORT">1883</entry>
    <entry key="AMQP_PORT">5672</entry>
    <entry key="BROKER_ID">rabbitmq</entry>
    <entry key="BROKER_PW">rabbitmq</entry>
    <entry key="MQTT_VirtialHost">/</entry>
    <entry key="QUEUE_NAME_CARD">atis-mybi</entry>
    <entry key="QUEUE_NAME_BIS">atis-bis</entry>
    <entry key="QUEUE_NAME_DSRC">atis-dsrc</entry>
    <entry key="QUEUE_NAME_AVI">atis-avi</entry>
    <entry key="USER_INBOXES_EXCHANGE">amq.topic</entry>
    <entry key="MESSAGE_CONTENT_TYPE">application/vnd.ccm.pmsg.v1+json</entry>
</properties>
```

[그림 19] XML 설정

3.4.4 SQL 쿼리

MS-SQL DBMS에 SQL 쿼리를 전달하기 위한 SQL 쿼리는 [표 6]과 같다.

[표 6] SQL 쿼리

쿼리
SELCET * FROM DATABASE.TARGET_TABLE WHERE DATE ORDER BY DATE

SQL의 구조는 간단하다. 교통 데이터 타입 별 저장 되어있는 테이블에서 해당 날짜를 기준으로 모든 컬럼을 가지고 온 후, 날짜의 오름차순으로 데이터를 정렬하여 테이블 형식으로 가져오는 쿼리이다.

3.4.5 멀티스레드

본 연구에서 시뮬레이터는 멀티스레드 기반으로 설계되었다. 스레드란 어떠한 프로그램 내에서 프로세스가 실행되는 흐름의 단위를 말한다. 일반적으로 한 프로그램은 하나의 스레드를 가지고 있지만, 프로그램 환경에 따라 둘 이상의 스레드를 동시에 실행할 수 있다. 이러한 실행 방식을 멀티스레드(Multithread)라고 한다. 웹 어플리케이션은 교통 데이터의 종류에 따라 각각의 스레드(Thread)가 독립적으로 실행되도록 멀티스레드(MultiThread) 방식으로 설계 및 구현하였다.

본 연구에서 사용된 멀티스레드 코드 일부를 소개한다. [그림 20]은 Controller 클래스 Card 교통데이터 타입의 스레드를 등록하는 코드 일부이다. [그림 21]은 Send 클래스에서 교통 데이터 종류에 따라 해당 Method가 실행되는 코드 일부이다.

```
/**
 * 스레드 등록
 * */
ExecutorService exService = Executors.newFixedThreadPool(fixedThreads);
ThreadPoolExecutor threadPoolExecutor = (ThreadPoolExecutor) exService;
Logger logger = Logger.getLogger(getClass());

try {
    if (chkData.isOkCARD() && flagValue == true) {

        logger.info("[Post] isOKCARD input");
        Send sendCard_thread = new Send();

        sendCard_thread.type_parameter("CARD");
        sendCard_thread.txtDate_parameter(txtDate);
        sendCard_thread.txtTimeInterval_parameter(txtTimeInterval);

        exService.submit(sendCard_thread);
        logger.info("Card_thread input queue...");

    }
```

[그림 20] 스레드를 등록하는 코드

```
@Override
public String call() throws MqttException, InterruptedException {
    switch (type) {
    case "CARD":
        try {
            sendCardMessage(txtDate, txtTimeInterval);

        } catch (IOException | TimeoutException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        break;

    case "BIS":
        try {
            sendBisMessage(txtDate, txtTimeInterval);
        } catch (IOException | TimeoutException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
```

[그림 21] 교통 데이터 종류에 따라 Method가 동작하는 코드

3.4.6 Publish/Subscribe

MS-SQL 서버에서 전달 받은 데이터를 MQTT Topic 설계에 따라 Publisher는 MQTT 브로커 서버에 Publish 해야 한다. 또한 본 연구에서 Subscriber인 Kafka Broker는 관심 있는 Topic을 Subscribe 해야 한다. 이에 따라 Publish와 Subscribe와 관련된 주요 Method를 살펴본다. MQTT 클라이언트 소스는 Paho 오픈 소스 라이브러리를 활용하였다.

(1) Connection Set Method

[그림 22]는 MQTT Broker의 Connection 하기 위해 옵션을 설정하고, Set 하는 Method이다.

```
private void setConOpts(MqttConnectOptions conOpts) {
    // provide authentication if the broker needs it
    conOpts.setUserName(properties.getProperty("BROKER_ID"));
    conOpts.setPassword(properties.getProperty("BROKER_PW").toCharArray());
    conOpts.setCleanSession(true);
    conOpts.setKeepAliveInterval(60);
}
```

[그림 22] Connection Method

(2) MQTT Setup Method

[그림 23]은 MQTT 기본 셋업과 관련한 Method이다.

```
public void setUpMqtt() throws MqttException {
    clientId = getClass().getSimpleName() + ((int) (10000 * Math.random()));
    client = new MqttClient(brokerUrl, clientId);
    conOpt = new MyConnOpts();
    setConOpts(conOpt);
    receivedMessages = new ArrayList<MqttMessage>();
    expectConnectionFailure = false;
}
```

[그림 23] MQTT Setup Method

(3) Publish Method

[그림 24]는 Client와, Topic, QoS를 입력 받아 Publish 하는 Method이다.

```
private void publish(MqttClient client, String topicName, int qos,
        byte[] payload) throws MqttException {
    MqttTopic topic = client.getTopic(topicName);
    MqttMessage message = new MqttMessage(payload);
    message.setQos(qos);
    MqttDeliveryToken token = topic.publish(message);
    token.waitForCompletion();
}
```

[그림 24] Publish Method

(4) Subscribe Method

[그림 25]는 생성된 Client를 기반으로 하여, MQTT Broker에 Connect하고, Subscribe하는 Method이다.

```
public void testSubscribeQos1() throws MqttException, InterruptedException {
    System.err.println("sub");
    client.connect(conOpt);
    client.setCallback(this);
    client.subscribe(topic, 1);


    try {
        System.in.read();
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    client.disconnect();

}
```

[그림 25] Subscribe Method

[그림 26]은 위의 코드를 바탕으로 Publish/Subscribe 하는 동안 RabbitMQ 플러그인의 메인 화면이다.

[그림 26] RabbitMQ Main 화면

Ⅳ. 시뮬레이터 성능 평가

본 장에서는 성능 평가를 위한 시스템 환경에 대해 4.1절에서 설명하고, 시뮬레이터의 실행 시간과 메시지 전송률과 관련하여 성능을 평가를 4.2절에서 설명하고, 4.3절에서 테스트를 요약한다.

4.1 시스템 환경 구축

본 논문에서는 웹에서 사용자가 시뮬레이터 옵션을 선택하고, 실제 교통데이터를 기반으로 성능평가가 진행이 되며, 아래 표는 본 실험에서 사용하는 시스템은 개발 환경을 나타낸 것이다.

(1) MQTT Broker 서버

[표 7] MQTT Broker 사양

Broker	RabbitMQ Server 3.6.5
OS	Windows Serer 2012 R2
CPU	Intel(R) Xeon(R) CPU E3-1270 @ 3.50GHz
RAM	32GB
DISK	HDD 2TB

(2) MS-SQL 서버

[표 8] MS-SQL 서버 사양

MS-SQL	Microsoft SQL Server 2012
OS	Windows Server 2012 R2
CPU	Intel(R) Xeon(R) CPU E3-1270 v3 @ 3.50GHz
RAM	16GB
DISK	HDD 4TB

(3) 웹 서버

[표 9] 웹 서버 사양

OS	Windows 10 Pro
CPU	Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
RAM	16GB
DISK	HDD 2TB

(4) Kafka 서버

[표 10] Kafka 서버 사양

OS	CentOS 6.6
CPU	Xeon E5-2630 v2<2.60GHz>
RAM	128GB
DISK	SSD 500GB

4.2 성능 평가

본 실험에서는 실제 교통 데이터를 기반으로 MQTT QoS Level 1로 고정하여 시뮬레이터의 실행시간과 메시지 전송률에 초점을 맞추어 성능 평가를 한다.

4.2.1 시뮬레이터 실행 시간

본 실험에서는 웹 실행 시점 즉, 사용자가 시뮬레이터 옵션을 선택하고 Start 버튼을 누른 시점과 MQTT Broker가 첫 메시지를 받은 시점을 ms 단위로 측정하여, 두 시점의 차이를 계산하여 실행시간을 측정한다.

조건 1. 시뮬레이터의 최대 비용을 기준으로 실험을 설계한다.
조건 2. 조건 1의 부합하는 교통 데이터 타입을 선택한다.
조건 3. 시간 간격은 0으로 한다.(시간복잡도 N)
조건 4. 같은 날짜를 선택하고, 총 10회 반복한다.

본 실험에서 실행 시점과 관련된 비용을 살펴보면 MS-SQL Server 커넥션 비용, MS-SQL Server에서 데이터를 정렬하여 받아오는 비용, 받아온 데이터를 HashMap에 임시저장 하는 비용, MQTT Broker 커넥션 비용, HashMap에 저장된 데이터를 JSON 파일 포맷으로 변환하고 Publish 하는 비용이 있다.

조건 1, 2를 바탕으로 교통 데이터 종류 중 하루 데이터 적재량이 가장 많

은 DSRC 데이터를 선택한다. DSRC 데이터는 2.1 절에서 앞서 소개한대로 하루 데이터 건수가 약 200만 건이다. 조건3, 4에 맞춰서 시뮬레이터 옵션을 선택한 후 실험을 한다. 위 조건을 바탕으로 실험한 결과는 [표 11]과 같다.

[표 11] 시뮬레이터 실행 시간

시뮬레이터 실행 시간										
측정	1회	2회	3회	4회	5회	6회	7회	8회	9회	10회
실행 시간 (ms)	7.226	7.116	8.29	7.465	7.435	7.403	7.44	8.624	7.052	7.514
평균 (ms)	7.5565									

IoT 시뮬레이터 실행 시간의 최대값은 8.624ms이며, 최소값은 7.052ms이다. 10회 평균 7.5565ms로 측정되었다. 간헐적으로 평균과 차이가 있는 수치도 존재하지만, 시뮬레이터의 실행 시간이 평균에 가깝게 몰려있음을 알 수 있다.

4.2.2 메시지 전송률 측정

본 실험에서는 단일 컴퓨팅 환경에서 시뮬레이터의 멀티스레드 기능을 이용하여, Publisher의 개수를 1~4 까지 늘려서 시간 간격에 따른 초당 메시지 전송량을 측정하고자 한다.

조건 1. Publisher 개수를 1~4까지 점차적으로 늘려서 측정한다.

조건 2. 시뮬레이터 옵션의 시간 간격을 0.1초, 0.05초, 0.01초, 0초(시간복잡도 N)로 단계별로 평균 메시지 전송률을 측정한다.

조건 3. 하루 적재량이 가장 많은 DSRC 데이터를 선택한다.

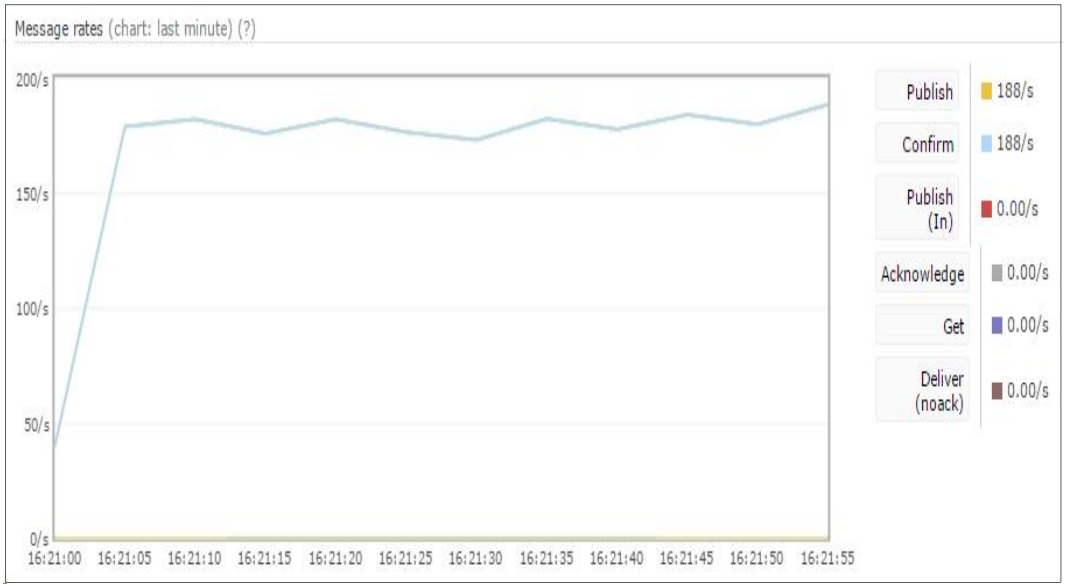
조건 4. 측정의 정확성을 위해 같은 날짜를 선택한다.

조건 5. 초당 메시지 전송률과 에러율을 측정한다.

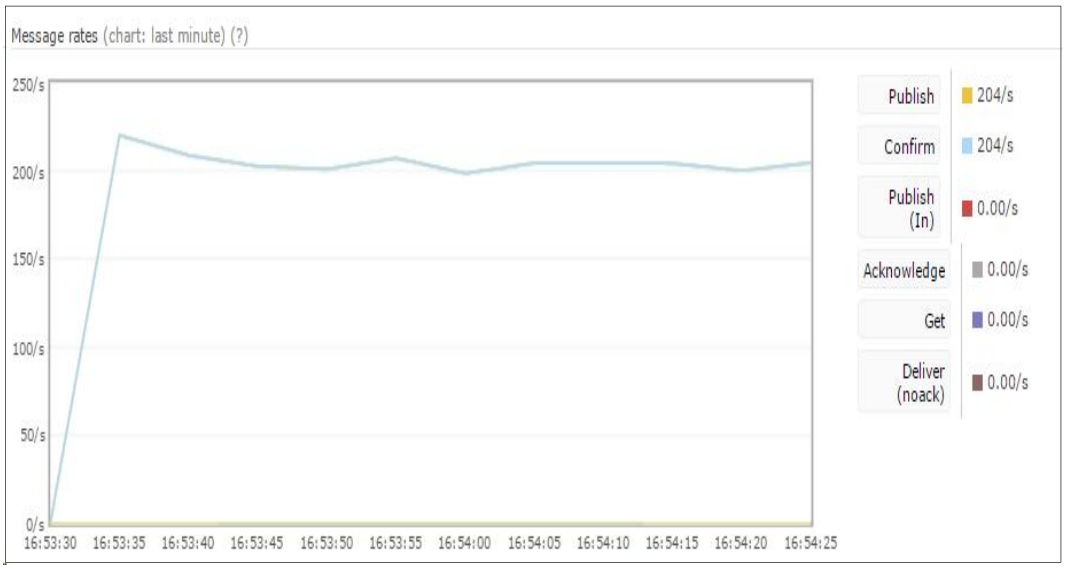
본 실험에서의 메시지 전송률의 영향을 끼치는 비용을 살펴보면 MQTT Broker 커넥션 비용, HashMap에 저장된 데이터를 JSON 파일 포맷으로 변환하고 Publish 하는 비용이 있다. 위 조건을 바탕으로 아래 [그림 27~30]은 Publisher 1~4까지의 시간 복잡도 N의 대한 메시지 전송률을 RabbitMQ Plugin에서 측정된 결과를 나타낸다. [그림 27]을 살펴보면 Publish 1개에서 초당 평균 120개의 메시지를 전송하는 것을 볼 수 있다. 그림 28~30도 비슷한 내용이므로 설명을 생략한다.

[그림 27] Publisher 1개, 시간복잡도 N

[그림 28] Publisher 2개, 시간복잡도 N



[그림 29] Publisher 3개, 시간복잡도 N



[그림 30] Publisher 4개, 시간복잡도 N

측정된 메시지 전송률을 요약하면 아래 [표 12~15]와 같다. Publisher 개수의 증가에 따라 시간간격을 조절하여, 성능테스트를 진행하였으며, 시간 간격 0.1, 0.5초일 때는 Publisher의 개수가 추가되어도, 비교적 비례적으로 메시지 전송률이 증가하지만, 0.01초, 0초 일 경우에는 메시지 전송률의 증가 값이 감소하는 것을 볼 수 있다. 즉, 초당 메시지 전송률이 100/s 부근부터 메시지 전송률의 증가 값이 감소하는 것을 볼 수 있다. 또한 Publisher 1개당 최대 메시지 전송률은 120/s 이며, 에러율은 모든 구간에서 0%이다.

[표 12] Publisher 1, 메시지 전송률

MQTT Broker (QoS 1)				
Publisher 1개				
Time-Interval	0.1초	0.05초	0.01초	0초 (시간복잡도 N)
에러율(%)	0	0	0	0
메시지 전송률(/sec)	9.2/s	17/s	52/s	120/s

[표 13] Publisher 2, 메시지 전송률

MQTT Broker (QoS 1)				
Publisher 2개				
Time-Interval	0.1초	0.05초	0.01초	0초 (시간복잡도 N)
에러율(%)	0	0	0	0
메시지 전송률(/sec)	19/s	34/s	97/s	160/s

[표 14] Publisher 3, 메시지 전송률

MQTT Broker (QoS 1)				
Publisher 3개				
Time-Interval	0.1초	0.05초	0.01초	0초 (시간복잡도 N)
에러율(%)	0	0	0	0
메시지 전송률(/sec)	28/s	49/s	136/s	188/s

[표 15] Publisher 4, 메시지 전송률

MQTT Broker (QoS 1)				
Publisher 4개				
Time-Interval	0.1초	0.05초	0.01초	0초 (시간복잡도 N)
에러율(%)	0	0	0	0
메시지 전송률(/sec)	37/s	67/s	167/s	204/s

4.3 성능 테스트 요약

본 연구에서 성능 평가는 IoT 시뮬레이터 실행시간과 메시지 전송률에 초점을 맞춰 진행되었으며, IoT 시뮬레이터 실행 시간의 최대값은 8.624ms, 최소값은 7.052ms, 10회 평균 7.5565ms로 측정되었다. 메시지 전송률 측정에서는 Publisher 1개일 때 평균 120/s, Publisher 2개일 때 160/s, Publisher 3개일 때 188/s, Publisher 4개일 때 204/s로 메시지 전송률의 증가 값이 감소하는 추세에 있는 것을 볼 수 있다. 모든 구간에서 에러율은 0% 이였다. 즉, 시뮬레이터는 하루 적재량 1000만 건 이상 커버할 수 있으며, DSRC 하루 적재량인 200만 건을 충분히 커버한다.

본 성능 평가의 한계점에 대해 살펴보면, 시뮬레이터의 실행 시간 성능평가는 SQL 쿼리 복잡도와 데이터 량의 따라 평균시간의 편차가 충분히 커질 가능성이 있으며, 메시지 전송률 성능평가는 시스템 환경에 따라 메시지 전송률의 편차가 충분히 커질 가능성이 있다.

Ⅴ. 시각화

본 장에서는 시뮬레이터를 통해 스트림 데이터를 Apache Kafka에서 구독한 후 메시지를 분산하여, Spark 스트림 어플리케이션으로부터 REST 형식으로 데이터를 전송 받는 웹 서버를 구축해 Vaadin을 활용하여 실시간 시각화 분석 하는 일부분을 간략하게 소개하고자 한다.

5.1 시각화의 분석 방법

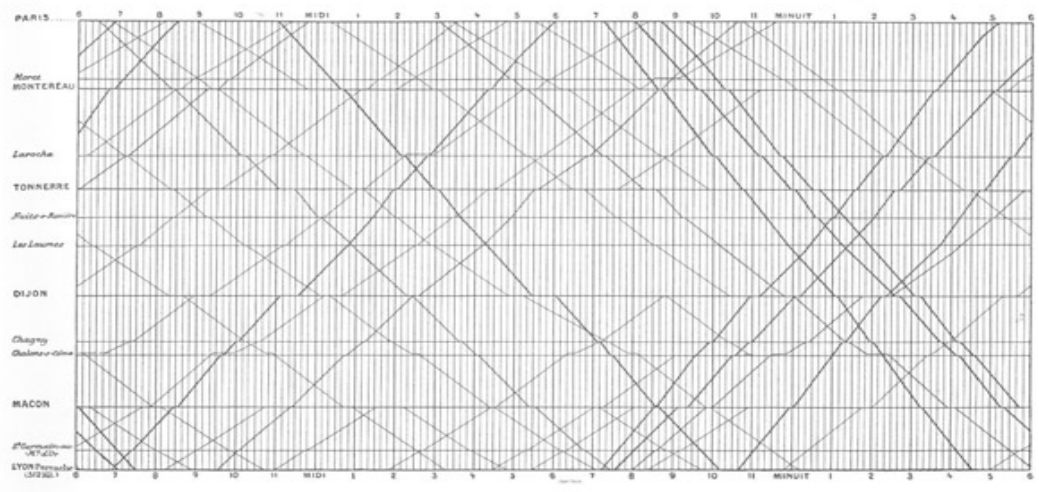
본 연구에서는 REST 웹 어플리케이션을 기반으로 시-공간 다이어그램인 마래 그래프(Marey Graph)와 Leaflet Map을 활용하여 실시간 버스 운행 모니터링을 시스템을 구현 하였다.

실시간 버스 운행 모니터링 시스템에서 활용되는 스트림 데이터는 버스운행기록(BIS)이며, 실시간 버스 운행 모니터링을 위하여 버스운행기록 RAW 데이터를 가공해야 하지만, 가공하여 전달하는 부분은 이전 단계인 Spark 스트림 어플리케이션에서 구현하는 부분이므로 생략하도록 한다.

5.1.1 Marey Graph

마래 그래프(Marey Graph)는 1885년에 에티엔느 쥘 마레(Etienne-Jules Marey)가 프랑스 파리에서 리옹으로 가는 열차의 스케줄 데이터를 이용하여 시각적으로 묘사하면서 등장했다. 마레 그래프는 하루 동안의 열차 경로를 시

공간 다이어그램을 통해 직관적으로 나타낼 수 있게 한다. 다시 말해서 하나의 이미지로 대중교통 시스템의 스케줄 정제할 수 있는 유용한 도구이며, 데이터 시각화 부문에서 혁신적인 디자인 사례로 자주 등장한다(송현진 2016).



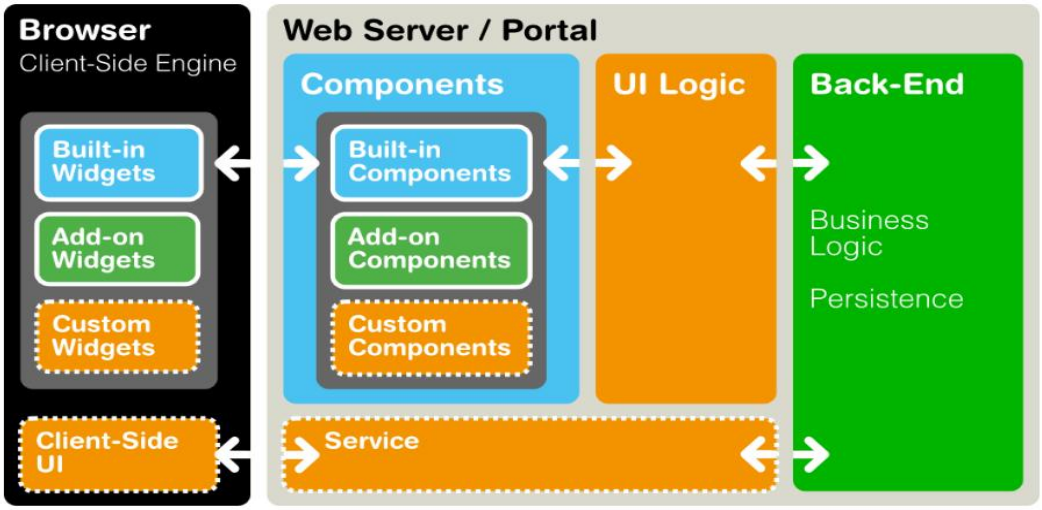
[그림 31] 파리의 열차 스케줄을 그린 최초의 마레 그래프

출처: http://www.c82.net/blog/?id=66

5.2 시각화 도구

5.2.1 Vaadin

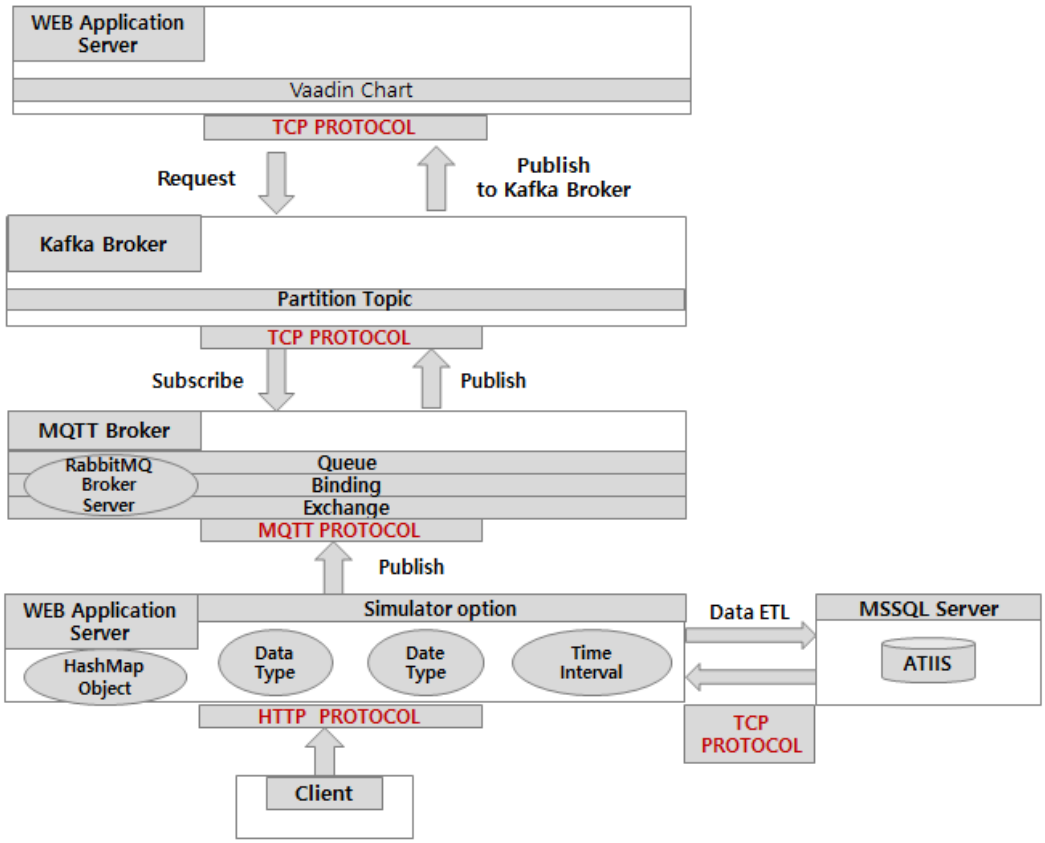
Vaadin은 GWT를 기본으로 확장한 Java기반의 리치 웹 프레임워크이다. 특징으로는 웹 UI를 보다 쉽게 제작할 수 있도록 도와주는 소스코드로 웹 UI를 제어할 수 있어서 레이아웃 템플릿과 컴포넌트 등을 간편하게 대시보드에 붙일 수 있도록 구성된다. [그림 32]는 Vaadin 어플리케이션 아키텍처를 나타낸다.



[그림 32] Vaadin 어플리케이션 아키텍처

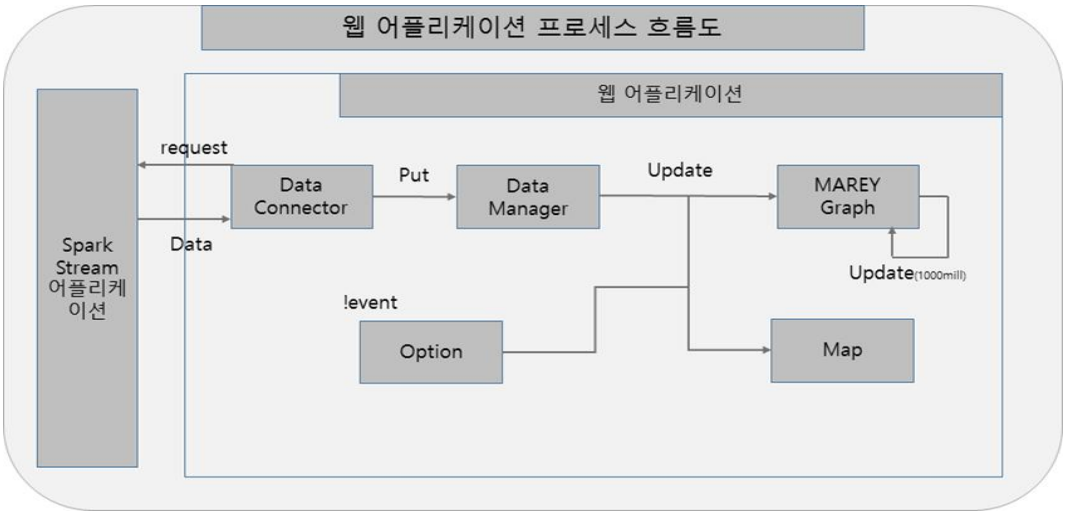
5.3 시각화 구현

REST 웹 어플리케이션을 기반으로 시각화 서비스를 구현하였고, 시각화 프레임 워크는 Vaadin을 사용하였다. 또한 교통 데이터는 버스정보기록(BIS)을 사용하였고, MAP은 오픈 소스 JavaScript library인 Leaflet MAP을 활용하였다. 아래 [그림 33]은 시스템의 전체적인 구성도이다. Kafka Broker에서 빅데이터 플랫폼으로의 저장, 관리는 본 연구에서는 다루지 않음으로 생략하였다.



[그림 33] 시각화 어플리케이션이 포함된 시스템 구성도

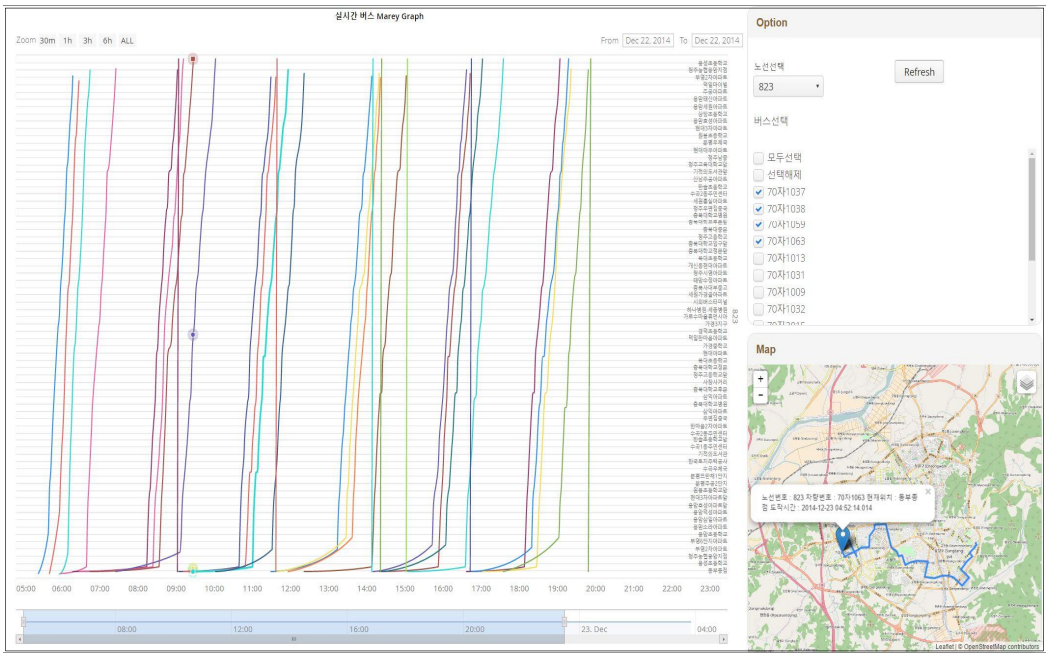
웹 어플리케이션의 프로세스 흐름도는 [그림 34]와 같다. 빅데이터 플랫폼인 SparkStream 어플리케이션에 데이터를 준 실시간으로 요청(Request)하여 데이터를 받아오고, Data Connector와 DataManager를 거쳐 옵션 이벤트에 따라 마래 그래프(Marey Graph)와 Map에 데이터를 디스플레이하는 방식이다.



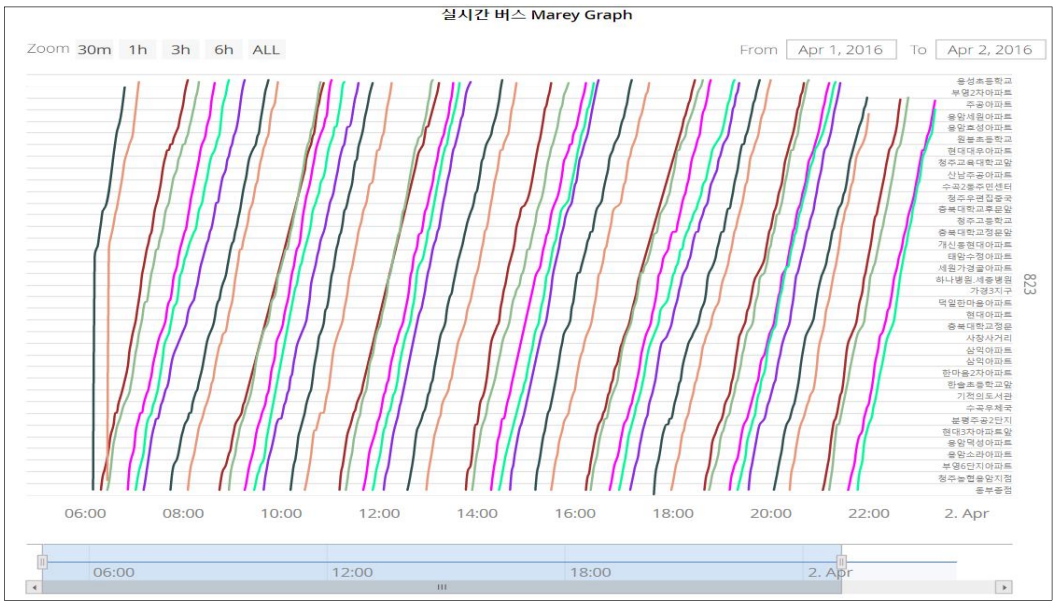
[그림 34] 시각화 웹 어플리케이션 프로세스 흐름도

웹 대시보드는 크게 마래 그래프 부분, MAP 부분, 시각화 도구 옵션 부분으로 [그림 35]처럼 구성되어있다. [그림 36]을 참조하여 보면, 마래 그래프에서 X축은 시간을 나타내고, Y축은 버스 정류장을 나타내며, 그래프는 노선의 버스를 의미한다. MAP은 [그림 37]처럼 버스 노선의 순서에 따라 정류장 좌표를 연결해 노선도를 그려주었다. 시각화 도구 옵션 [그림 38]처럼 노선과 해당 버스를 선택하여, 마래 그래프 또는 MAP에 그려지도록 하는 옵션이다.

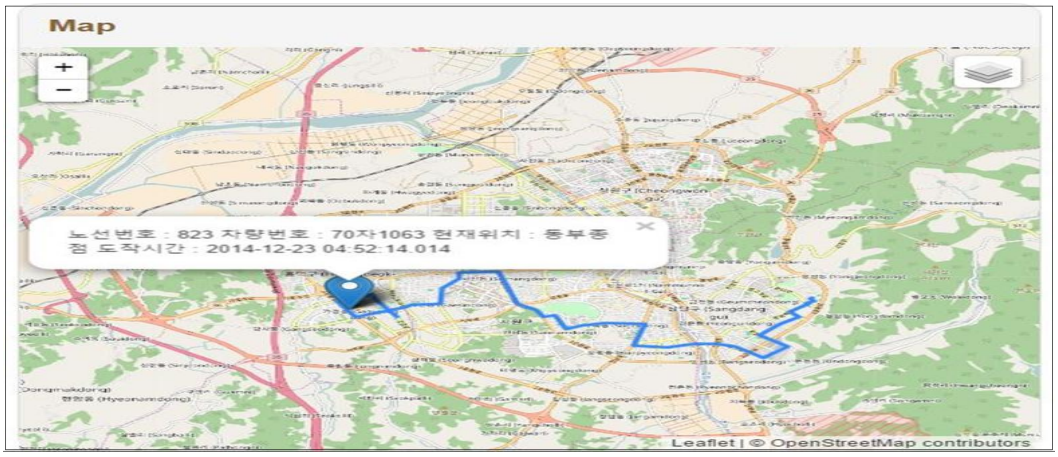
시각화 도구 옵션에서 버스 노선을 선택하고, 버스를 선택하면 앞선 시뮬레이터의 시간 간격의 맞춰 실제 데이터가 디스플레이된다. 마래 그래프는 버스가 정류장에 도착할 때마다 아래에서 위로 그려지며 각 선들은 노선을 운행한 차량 한 대씩 구분하여 나타낸 것이다. MAP에는 버스의 위치 좌표를 이용하여, 정류장에 도착할 때마다 버스의 위치가 이동한다.



[그림 35] 웹 대시보드



[그림 36] MAREY 그래프



[그림 37] 실시간 버스 위치정보

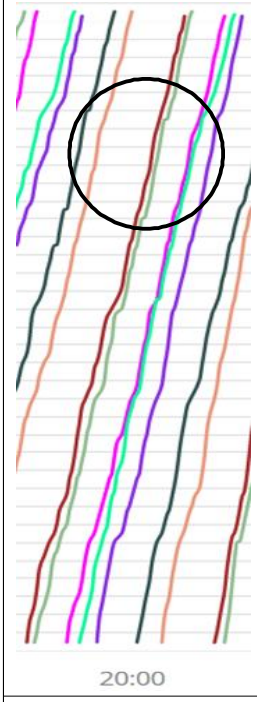
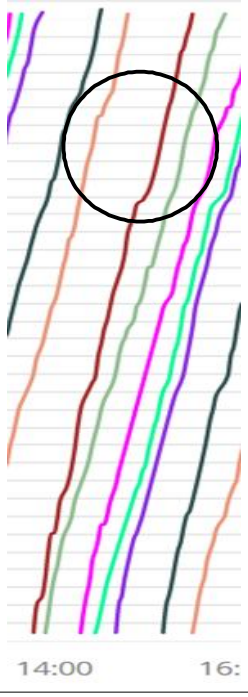
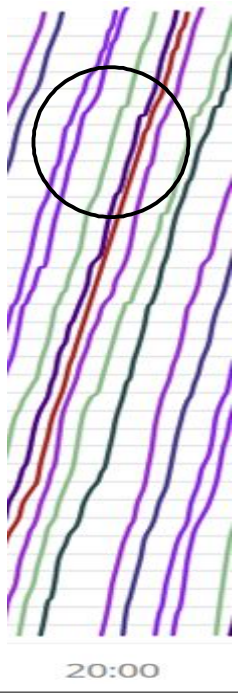
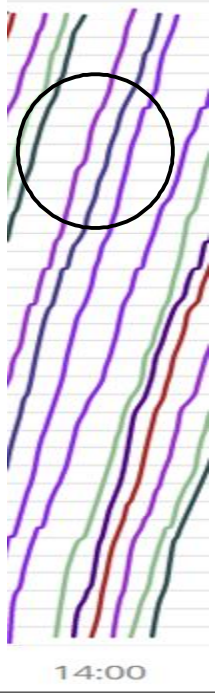
[그림 38] 시각화 도구 옵션

5.4 시각화 분석

청주시 교통 데이터 중 버스운행 기록 데이터(BIS)를 이용하여 실시간 버스 운행 스케줄 모니터링 시뮬레이션을 적용해 본 결과 몇 가지 운행특성을 찾을 수 있었다. [표 16]은 823번 노선의 2016년 4월 1일, 2일 운행기록 일부를 시

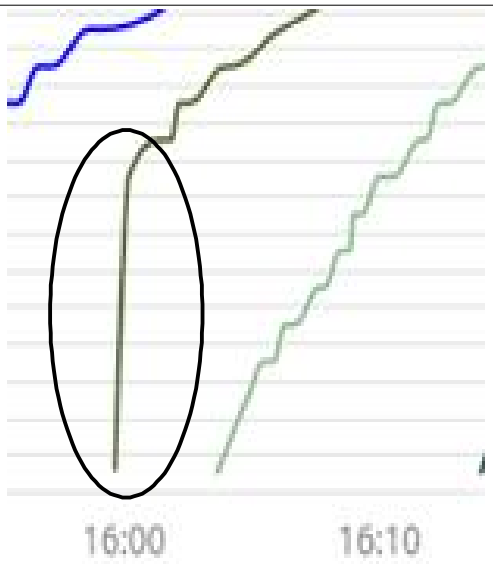
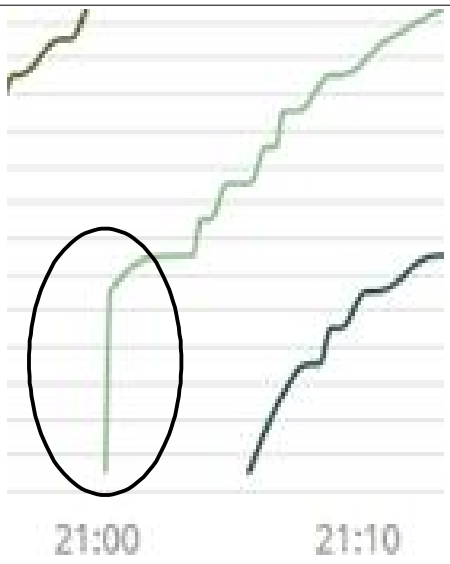
각화한 것이다. 첫 번째 특성은 개신동현대아파트, 충북대학교 정문 앞, 청주고등학교, 충북대학교 후문 앞 구간을 다른 시간과 비교했을 때, 퇴근시간대인 20시 쯤 그래프의 간격이 일정하지 않은 것을 볼 수 있다. 즉, 각 회차별로 배차간격을 기준으로 출발시간이 다름에도 불구하고 버스가 거의 동시에 정차하는 등의 불안정적인 배차간격을 찾을 수 있었다.

[표 16] 823번 노선의 배차간격 특성 파악 결과

<table>
<tr><th colspan="4">1. 노선의 배차간격 특성 파악</th></tr>
<tr><td colspan="4">823번 순환
개신동현대아파트 -> 충북대학교정문앞 -> 청주고등학교->충북대학교후문앞</td></tr>
<tr><td colspan="2">2016년 4월 1일</td><td colspan="2">2016년 4월 2일</td></tr>
<tr><td> 
</td><td> 
</td><td> 
</td><td> 
</td></tr>
<tr><td>퇴근시간</td><td>그 외 시간</td><td>퇴근 시간</td><td>그 외 시간</td></tr>
</table>

두 번째 특성으로 [표 17]을 살펴보면 823번 노선의 2016년 4월 1일 16시 동부종점~ 용암삼일아파트 구간, 21시 동부종점~ 부영6단지아파트 구간에서 그래프가 일직선으로 그려진 것을 볼 수 있다. 정확한 이유는 알 수 없으나 추정해 볼 수 있는 가능성으로는 첫째, BIS 관련 운행단말기의 문제일 가능성 둘째, 기점을 기준으로 그래프가 아래와 같이 그려지는 것을 보아, 배차를 시작하였으나 손님이 없어 버스의 운행단말기를 켜지 않고 운행하다가 손님이 타는 정류장에서 버스 운행단말기를 키는 경우를 생각해볼 수 있을 것 같다. 운행기기를 켜지 않으면 데이터가 발생하지 않기 때문이다.

[표 17] 823번 노선의 운행특이점

<table>
<tr><th colspan="2">2. 노선의 운행 특이 사항</th></tr>
<tr><td colspan="2">823번 순환</td></tr>
<tr><td>동부종점 -> 용성초등학교 ->,
청주농협용암지점 -> 부영2차아파트 ->
부영6단지아파트 -> 용암초등학교 ->
용암소라아파트 -> 용암삼일아파트</td><td>동부종점 -> 용성초등학교 ->
청주농협용암지점 -> 부영2차아파트 ->
부영6단지아파트</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>2016년 4월 1일 16시</td><td>2016년 4월 1일 21시</td></tr>
</table>

Ⅵ. 결론

IoT 기반 교통 스트림 데이터는 수집하는 디바이스의 종류가 증가함에 따라 실시간 데이터의 양도 매년 증가한다. 이러한 특성 때문에 기존에 교통 운영시스템에서 저장, 관리, 분석하기에는 어려움이 있다. 따라서 빅데이터 환경의 교통 운영시스템으로 전환이 필요하지만 급격한 전환은 기존 교통 운영시스템에 영향을 미치는 등 위험성이 존재한다. 이러한 이유로 기존 교통 운영시스템에 지장을 주지 않고, 실제 교통 데이터를 기반으로 빅데이터 플랫폼의 안정성, 실시간 분석 등의 테스트를 하기 위해 IoT 시뮬레이터가 필요하다.

본 연구에서는 IoT 대용량 메시지 전달 프로토콜인 MQTT를 바탕으로 청주시 실제 교통 데이터를 사용하여 다양한 시나리오에 따라 시뮬레이션을 할 수 있도록 IoT 시뮬레이터를 설계 및 구현하였다. 또한, 시뮬레이터의 실행 시간, 초당 메시지 전송량을 측정하여 자체 성능 평가를 진행하였고, 성능평가 결과 시뮬레이터의 실행 시간은 평균 7초대가 나왔으며, 단일 머신에서 메시지 전송률은 최대 120/s의 전송률을 보여주었다. 즉, 시뮬레이터는 하루 적재량 1000만 건 이상 커버할 수 있으며, DSRC 하루 적재량인 200만 건을 충분히 커버한다. 마지막으로 본 연구에서 구현한 시뮬레이터를 응용하여 스트림 데이터 시각화 및 분석을 통하여, 버스 운행 특성의 여러 패턴을 찾을 수 있었다. 첫째, 823번 노선의 퇴근시간대에 개신동현대아파트 ~ 충북대학교 후문 앞 까지 불안정적인 배차간격을 찾을 수 있었다. 둘째, 823번 노선의 2016년 4월 1일 16시 동부종점~ 용암삼일아파트 구간, 21시 동부종점~ 부영6단지아파트 구간에서 비정상적인 그래프가 그려지는 것을 찾을 수 있었다.

본 논문을 통해 실제 교통 데이터를 다양한 시나리오에 따라 시뮬레이션 할 수 있도록 IoT 시뮬레이터를 설계 및 구현했다는 점에서 의미가 있다. 이는 교통 분야뿐만 아니라 제조·의료 분야 등 비슷한 이유로 시뮬레이션이 필요할 경우 응용 할 수 있다는 점에서 기여할 수 있을 것으로 보인다. 또한 시각화 분석에 따라 대중교통 서비스 향상 등의 버스 운영정책이나 규정에 기여할 수 있을 것으로 보인다.

그러나 본 논문은 다음과 같은 한계점이 있으며 이들은 향후 과제로 남겨두었다. 첫 번째는 시뮬레이터의 성능평가에서 실행시간은 SQL 쿼리 복잡도와 데이터 량의 따라 평균시간의 편차가 충분히 커질 가능성이 있다는 점이다. 때문에 다양한 환경에서 수많은 테스트와 실험을 거쳐 실행 시간을 측정할 필요가 있다.

두 번째는 데이터의 메시지 전송률 성능평가는 표준화된 방식이 아니라는 점이다. 그렇기 때문에, 시스템 환경에 따라 메시지 전송률의 편차가 충분히 커질 가능성이 있다. 향후 과제로써 전문적인 벤치마크로 성능을 평가하여 객관성을 부여 할 필요가 있다.

세 번째 시각화 분석 시 다양한 기능을 추가 할 필요가 있다는 점이다. 현재 구현한 시각화는 마래 그래프(Marey Graph)와 Leaflet Map을 통하여 버스의 배차 간격과 버스의 위치 추적 기능을 제공한다. 향후 과제로써 스트림 데이터마이닝 알고리즘을 적용하여 다각화된 분석이 가능하게끔 기능을 추가 할 필요가 있다.

참 고 문 헌

<국내문헌>

정덕원 (2014), “교통 빅데이터의 실시간 분석 및 예측 서비스 프레임워크”, 건국대학교 대학원 박사학위 논문

이석주 외 (2013), “빅데이터를 이용한 교통정책 개발 및 활용성 증대방안”, 한국교통연구원

김상호 (2015), “Apache Spark를 활용한 교통빅데이터 분석 및 처리성능 평가”, 충북대학교 대학원 석사학위 논문

김상현 (2016), “안정적인 사물인터넷 플랫폼을 위한 MQTT 프로토콜 기반 데이터 수집 솔루션에 관한 연구”, 한밭대학교 정보통신전문대학원 석사학위 논문

송현진 (2016), “Marey 그래프를 이용한 대중교통 운행 스케줄 모니터링”, 충북대학교 대학원 석사학위 논문

<국외문헌>

M. Prihodko (2012), “Energy consumption in location sharing protocols for Android applications”, Master’s Thesis of Link opings University

<웹사이트>

The MQTT protocol (2016), http://www.mqtt.org/

MQTT Tutorial(2016), http://www.joinc.co.kr/w/man/12/MQTT/Tutorial

Introduction to the Spring Framework, http://docs.spring.io/spring-framewo

rk/docs/current/spring-framework-reference/html/overview.html#overvie
w-modules

실시간 교통빅데이터 시뮬레이션을 위한 IoT 시뮬레이터 설계 및 구현

안 상 희

충 북 대 학 교 대 학 원
빅데이터협동과정 빅데이터전공

(지도교수 조 완 섭)

요 약

IoT 기반 교통 스트림 데이터는 수집하는 디바이스의 종류가 증가함에 따라 실시간 데이터의 양도 매년 증가한다. 이러한 특성 때문에 기존에 교통 운영시스템에서 저장, 관리, 분석하기에는 어려움이 있다. 따라서 빅데이터 환경의 교통 운영시스템으로 전환이 필요하지만 급격한 전환은 기존 교통 운영시스템에 영향을 미치는 등 위험성이 존재한다. 이러한 이유로 기존 교통 운영시스템에 지장을 주지 않고, 실제 교통 데이터를 기반으로 빅데이터 플랫폼의

안정성, 실시간 분석 등의 테스트를 하기 위해 IoT 시뮬레이터가 필요하다.

본 연구는 IoT 프로토콜인 MQTT를 바탕으로 웹 기반 IoT 시뮬레이터를 구현하고, 청주시 실제 교통 데이터를 사용하여 시뮬레이션을 할 수 있도록 설계하였다. 시뮬레이터의 실행 시간, 초당 메시지 전송량을 측정하여 자체 성능 평가를 진행하였다. 성능평가 결과 IoT 시뮬레이터는 초당 120개의 메시지를 생성하여 전송하며 하루에 1000만개의 DSRC 메시지를 처리할 수 있어 청주시의 충분한 기능을 제공한다. 또한, 본 연구에서 구현한 시뮬레이터를 응용하여 스트림 데이터 시각화와 분석을 통하여, 버스 운행 특성의 여러 패턴을 찾을 수 있었다.

주요어 : 교통 빅데이터, 시뮬레이터, IoT, MQTT, 시각화