아파치 카프카의 메시지 지연시간 기반 로드 쉐딩 메커니즘

김하진*, 방지원*, 손시운*, 최미정*, 문양세*
*강원대학교 컴퓨터과학과
e-mail : {hajinkim, jiwonbang, ssw5176, mjchoi, ysmoon}@kangwon.ac.kr

Message Latency-based Load Shedding Mechanism in Apache Kafka

Hajin Kim*, Jiwon Bang*, Siwoon Son*, Mi-Jung Choi*, Yang-Sae Moon*
*Dept. of Computer Science, Kangwon National University

요 약

아파치 카프카(Apache Kafka)는 데이터 스트림을 실시간 전달하는 분산 메시지 큐잉 플랫폼이다. 카프카는 대다수의 실시간 처리 응용에 사용되는데, 흔히 데이터 스트림의 발생지와 실시간 처리 시스템 사이(입력) 또는 실시간 처리 시스템과 처리 결과의 목적지 사이(출력)에 배치된다. 분산 기술을 도입한 카프카는 다른 메시지 큐잉 기술에 비해 대용량 데이터 스트림을 더욱 빠르게 전달할 수 있다는 장점을 갖는다. 하지만, 카프카에 적재되는 데이터 스트림의 양과 실시간 처리 응용의 수가 증가할수록 메시지 지연시간은 매우 높아질 수 밖에 없다. 본 논문은 이러한 카프카의 메시지 지연 문제를 해결하고자 카프카의 로드 쉐딩 엔진을 제안한다. 로드 쉐딩의 세 가지 필수적인 결정에 따라, 제안하는 로드 쉐딩 엔진은 카프카의 프로듀서에서 지연시간이 기준치를 초과할 경우 일부 메시지 전송을 제한하여 지연시간을 줄인다. 실제 실시간 처리 응용으로 실험한 결과, 단일/다중 데이터 스트림 모두 로드 쉐딩이 바르게 작동하여 지연시간이 지속적으로 증가하지 않고 오르내림이 반복되는 추세를 보였다. 본 연구는 데이터 스트림의 입출력을 카프카로 관리하는 실시간 처리 응용에 로드 쉐딩 기법을 적용한 첫 번째 시도로서, 앞으로 데이터 스트림 처리에 사용될 의미 있는 연구라 사료된다.

1. 서론

데이터 스트림은 끊임없이 생성되는 연속된 형태의 데이터로서, 이를 수집 즉시 분석하기 위해서는 실시간 처리가 필수적이다[1, 2]. 또한, 데이터 스트림은 유입속도와 크기가 동적으로 변하기 때문에, 실시간 처리 시스템은 데이터 스트림을 안정적으로 처리할 수 있어야 한다. 특히, 데이터 스트림이 폭증하여 실시간 처리 시스템이 과부하 상태가 되면, 메시지 처리 지연시간이 증가하며 심할 경우 시스템이 중단될 수 있다. 이러한 문제를 해결하기 위해, 대다수의 실시간 처리 시스템에서는 입출력으로 메시지 큐잉 시스템을 도입한다.

아파치 카프카(Apache Kafka)[3, 4]는 다수의 서버를 통해 대용량 데이터 스트림을 빠르게 전달하는 분산 메시지 큐잉 플랫폼으로 기존 메시징 시스템과 달리 확장성과 고가용성을 보장하고, 분산 배치 처리 기법을 사용하여 성능이 매우 좋은 특징이 있다[5]. 하지만, 카프카에서도 데이터 스트림의 양이 많아질수록 또는 실시간 처리 시스템의 작업 시간이 길어 구독(subscribe) 주기가 길어질수록, 카프카에 적재된 메시지의 지연시간(latency)이 길어진다. 이러한 상황이 지속될 경우, 카프카에는 많은 메시지가 대기하며 데이터 스트림의 실시간성을 보장할 수 없게 된다. 따라서 본 논문에서는 실시간 처리 시스템의 입출력이 카프카로 구성된 실시간 처리 응용에 로드 쉐딩 메커니즘을 적용한다. 로드 쉐딩[6, 7]이란 데이터 스트림이 폭증하여 시스템의 처리 능력을 초과하는 경우, 시스템이 과부하 상태가 되지 않도록 처리되지 않은 일부 메시지(튜플)을 버리는 기법이다. 로드 쉐딩 기법을 사용하는 경우, (1) 로드 쉐딩 시기 결정, (2) 로드 쉐딩의 위치 결정, (3) 로드 쉐딩의 수량 결정의 세 가지 요소를 필수로 판단해야 한다[6]. 제안하는 로드 쉐딩 엔진에서는 카프카에 로드 쉐딩을 도입하기 위해, 카프카의 (2) 프로듀서(producer)에서 (1) 지연시간이 기준치를 초과할 경우 (3) 일부 메시지 전송을 제한한다. 여기서 프로듀서는 실시간 처리 시스템의 입력을 의미하며, 지연시간의 임계치와 메시지 전송 비율은 데이터의 특성을 알고 있는 시스템 관리자가 결정한다.

실험에서는 제안하는 로드 쉐딩 엔진이 잘 동작함을 확인하기 위해, 실시간 처리 응용으로 아파치 스톰(Apache Storm)[8, 9]을 사용한다. 스톰은 분산 환경에서 데이터 스트림을 처리하는 실시간 분산 데이터 처리 플랫폼으로, 토폴로지라는 미리 정의된 알고리즘을 끊임없이 수행한다. 실험에 사용한 토폴로지는 입력된 배열을 정렬하여 반환한다. 실험 결과, 로드 쉐딩을 적용하지 않은 기존 시스템의 경우에는 메시지 지연시간이 지속적으로 증가한 반면, 로드 쉐딩 엔진이 적용된 시스템의 경우에는 지연시간이 지속적으로 증가하지 않고 오르내림을 반복하며 안정적인 추세를 보였다. 또한, 데이터 스트림의 입력 소스가 여러 개인 경

* 이 논문은 2018 년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임 (No. 2016-0-00179, 데이터 스트림 정제를 위한 지능형 샘플링 및 필터링 기술 개발)

우에도 병렬적으로 각 소스의 로드 쉐딩을 잘 수행하였다. 본 연구는 데이터 스트림을 카프카로 전달하는 실시간 데이터 스트림 처리 응용에 로드 쉐딩 기법을 적용한 첫 번째 시도로서, 앞으로 대용량 데이터 스트림 처리에 사용될 의미 있는 연구라 사료된다.

2. 관련 연구

2.1. 아파치 카프카

아파치 카프카[3, 4]는 대용량 데이터 스트림을 빠르게 전달하는 분산 메시지 큐잉(queueing) 플랫폼이다. 카프카는 그림 1 과 같이 프로듀서(producer), 브로커(broker), 컨슈머(consumer)로 구성된다. 여기서 브로커는 확장성과 고가용성을 위해 하나 이상의 서버로 구성된 클러스터로 동작한다. 프로듀서는 카프카 클러스터에 메시지를 발행(publish)하는 애플리케이션이고, 컨슈머는 원하는 데이터를 구독(subscribe)하는 애플리케이션이다. 그리고 카프카 클러스터의 분산 관리를 위해 아파치 주키퍼(Apache Zookeeper)를 사용한다. 카프카에서 메시지는 토픽(topic)이라는 채널을 통해 공유된다. 즉, 토픽은 프로듀서가 보내는 데이터들의 카테고리이며 브로커에 저장된다. 다수의 프로듀서가 하나의 토픽에 메시지를 생산할 수 있고, 다수의 컨슈머가 하나의 토픽을 구독하여 데이터를 소비할 수도 있다.

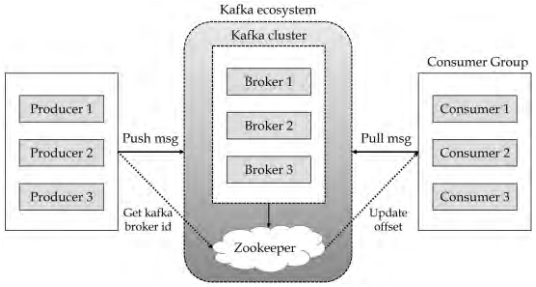

그림 1. 카프카 클러스터와 구성요소[10].

카프카는 다음과 같이 기존 메시지 큐잉 시스템과 차별화되는 주요 특징이 있다. 먼저, 카프카는 프로듀서와 컨슈머가 분리된 발행/구독 모델을 적용하기 때문에 확장성과 고가용성을 보장한다. 또한 여러 프로듀서와 여러 컨슈머가 하나의 토픽에 접근 가능한 구조이기 때문에 여러 소스에서 발생하는 데이터를 하나의 토픽에 저장할 수 있고, 하나의 토픽에 저장된 데이터를 다양한 용도로 사용할 수 있다. 다음으로, 카프카는 확장이 용이하고, 분산 배치 처리 기법을 사용해 매우 높은 성능을 가진다. 이런 장점들 때문에 카프카는 전체 데이터 처리 시스템의 중앙에서 모든 데이터 흐름을 관리하는 플랫폼으로 사용된다. 특히, 실시간 데이터 파이프라인 및 스트리밍 앱을 제작하는데 많이 사용된다. 본 논문에서는 실시간 데이터 스트림 처리 시스템에서 데이터 스트림의 입출력으로 카프카를 사용할 때 발생하는 처리 지연 문제를 해결한다.

2.2. 로드 쉐딩

로드 쉐딩[6, 7]이란 시스템에 유입되는 데이터 스트림의 유입 속도가 폭증하여 시스템의 처리 능력을 초과하는 경우에, 시스템이 과부하 상태가 되지 않도록 처리되지 않은 일부 메시지(튜플)을 버리는 기법이다. 입력 데이터가 폭증하여 시스템이 과부하 상태가 되면 시스템이 중단될 수 있기 때문에, 데이터의 양이 많거나 폭증의 가능성이 있을 경우에는 로드 쉐딩이 꼭 필요하다. 특히 데이터 스트림은 끊임없이 유입되고 유입속도와 크기가 동적으로 변화하기 때문에 이를 처리하는 DSMS(Data Stream Management System)[2]에 로드쉐딩이 주로 사용된다. 또한 데이터 스트림의 전송률뿐만 아니라 데이터 스트림의 크기, 종류, 질의 등에 따라 처리 시간에 지연이 발생할 수 있고, 이러한 경우 실시간 처리를 보장하기 위해 로드 쉐딩이 필요하다.

로드 쉐딩은 다음과 같이 세 가지의 필수 요소를 결정해야 한다[6].

- 로드 쉐딩의 시기 결정(determining when to shed load): 언제 로드 쉐딩을 할 지 결정하기 위해 시스템의 처리 한계 부하와 현재 시스템 부하를 계산해야 한다. 측정된 현재 시스템 부하와 처리 한계 부하를 비교하여 현재 시스템 부하가 처리 한계 부하를 초과하면 로드 쉐딩을 진행한다.
- 로드 쉐딩의 위치 결정(determining where to shed load): 로드 쉐딩은 시스템의 모든 단계에 들어갈 수 있기 때문에 데이터는 시스템의 어떤 지점에서도 데이터가 버려질 수 있다. 일반적으로 더 빨리 버리는 것이 시스템의 불필요한 처리를 막지만 다양한 질의들이 병렬 처리되는 경우 초기에 데이터를 버리면 많은 응용 프로그램에 영향을 줄 수 있다. 따라서 시스템의 처리 구조에 맞게 데이터를 버리는 지점을 결정해야 한다.
- 로드 쉐딩의 수량 결정(determining how much load to shed): 로드 쉐딩 동작 시 얼마나 버릴 것인지 비율을 결정해야 한다.

본 논문에서는 실시간 데이터 스트림 처리 시스템에서 카프카를 입출력으로 사용할 때 발생하는 지연 문제를 해결하는 지연시간 기반 로드 쉐딩 엔진을 제안한다.

3. 카프카 지연시간 기반 로드 쉐딩 엔진

본 절에서는 아파치 카프카를 실시간 처리 시스템의 입출력에 사용하는 응용에 적용한 로드 쉐딩 메커니즘을 설명한다. 제안하는 로드 쉐딩 엔진의 세 가지 필수 결정은 다음과 같이 설계한다. 첫째, 로드 쉐딩의 시기 결정은 메시지 처리 지연시간이 로드 쉐딩 기준 시간을 초과할 때로 설정한다. 사용자는 끊임없이 유입되는 데이터 스트림의 처리 결과를 실시간으로 모니터링하기를 원하기 때문에 메시지 처리의 실시간성이 무엇보다 중요하다. 데이터 스트림 유입 속도가 동적으로 변하는 만큼 실시간 처리 보장을 위해 시스템의 처리 지연시간이 일정 수준 이하로 유지되어야 한다. 따라서 제안하는 로드 쉐딩 엔진에서는 메시지 처리 지연시간을 측정하여 지연시간이 로드 쉐딩 임계치를 초과하면 로드 쉐딩을 시작한다. 둘째, 로드 쉐딩의 위치 결정은 데이터 스트림의 발생지와 실시간 처리 시스템 사이에서 데이터 스트림을 전달하는 카프카 프로듀서에서 수행하도록 설정한다. 이는 실시간 처리 시스템이 데이터 스트림을 처리하기 전에 메시지를 버려 불필요한 처리를 수행하지 않기 위함이다. 또한, 실시간 처리 응용과 데이터 스트림 입력을 분리하여, 제안하는 로드 쉐딩 엔진에 다양한 데이터 스트림 처리 응용을 적용할 수 있도록 하기 위함이다. 셋째, 로드 쉐딩의 수량 결정은 시스템 관리자가 설정한다. 이는 시스템 관리자의 인지하에 전체 메시지에서 실시간 처리 시스템에 전달할 로드 쉐딩 비율을 설정하여, 전체 시스템의 부하를 일정하게 관리하기 위함이다.

그림 2 는 제안하는 카프카 로드 쉐딩 엔진의 구조이다. 그림을 보면, 실시간 메시지 처리 시스템의 데이터 입출력은 카프카 프로듀서와 컨슈머로 구성된다. 그리고 아파치 스톰(Apache Storm)[8, 9], 아파치 스파크 스트리밍(Apache Spark Streaming)[11], 카프카 스트림즈(Kafka Streams)[12] 등 실시간 메시지 처리 시스템을 통해 데이터를 처리 및 분석한다. 로드 쉐딩 매니저를 간단히 설명하면, 로드 쉐딩 매니저는 실시간 메시지 처리 시스템과 독립적으로 동작하는데, 처리 완료된 메시지를 받는 카프카 컨슈머와 통신하여 로드 쉐딩 여부를 판단한다. 로드 쉐딩 매니저가 데이터베이스에 로드 쉐딩 여부를 저장하고, 카프카 프로듀서가

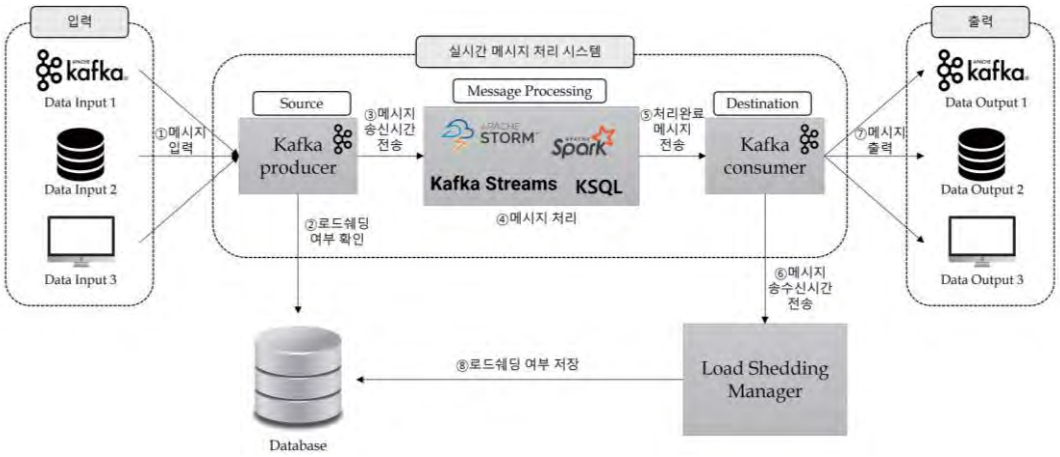

그림 2. 로드 쉐딩 메커니즘을 도입한 카프카 기반 실시간 처리 시스템의 구조.

주기적으로 데이터베이스를 읽어 로드 쉐딩 여부를 확인한다. 이때, 만약 로드 쉐딩을 수행해야 한다면 미리 설정된 로드 쉐딩 비율로 데이터를 버린다.

자세한 동작 절차는 다음과 같다. 먼저 카프카, 데이터베이스 등 다양한 데이터 스트림의 입력으로부터 메시지를 입력받는다(①). 다음으로, 카프카 프로듀서는 입력받은 메시지 송신시간을 붙여 실시간 메시지 처리 시스템으로 전달한다(③). 이때, 프로듀서는 로드 쉐딩 여부가 저장된 데이터베이스를 주기적으로 확인한다(②). 만약 로드 쉐딩이 동작하지 않는다면 그대로 메시지를 보내고, 로드 쉐딩이 동작 중이라면 로드 쉐딩 비율만큼의 메시지만 보낸다. 그리고 메시지 처리 플랫폼에서 받은 메시지를 처리하고(④) 컨슈머로 전송한다(⑤). 컨슈머는 메시지 송신시간과 수신 즉시 측정한 수신시간을 로드 쉐딩 매니저에게 보내고(⑥), 처리 완료된 메시지는 카프카, 데이터베이스 등 다양한 출력으로 보낸다(⑦). 메시지 송수신시간을 받은 로드 쉐딩 매니저는 그 차이를 처리 지연시간으로 계산하고, 로드 쉐딩 여부를 판단하여 데이터베이스에 저장한다(⑧).

제안하는 로드 쉐딩 엔진은 데이터 스트림의 소스 단위로 로드 쉐딩을 수행하여, 다중 소스의 경우도 동시에 병렬로 각 소스에 대한 로드 쉐딩을 지원한다. 이에 대한 자세한 동작 방식은 그림 3 과 같다. 먼저, 카프카 컨슈머가 로드 쉐딩 매니저에 소켓통신을 요청하면 로드 쉐딩 매니저는 요청을 수락하고(①), 연결된 소켓을 넘겨주어 작업 스레드를 생성한다(②). 통신이 연결되면 컨슈머가 작업 스레드에게 메시지 송수신시간을 전송한다(③). 송수신시간을 받은 각 작업 스레드는 처리 지연시간을 계산하여 각 소스의 로드 쉐딩 여부를 판단한다.

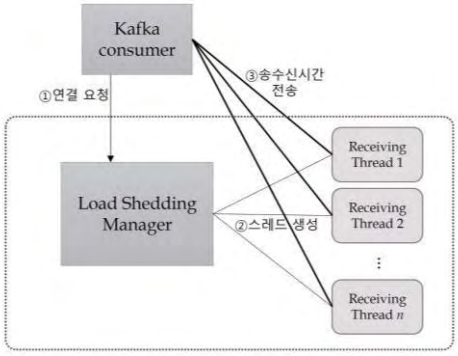

그림 3. 로드 쉐딩 매니저의 다중 소스 처리 절차.

그림 4 는 로드 쉐딩 매니저의 각 작업 스레드에서 동작하는 로드 쉐딩 판별 알고리즘이다. 로드 쉐딩 임계치 ε 과 컨슈머와 통신하는 소켓 *socket* 을 입력받아 소켓 연결이 끊어지기 전까지 반복한다. 먼저, 컨슈머로부터 메시지 송수신시간을 받으면 송신시간과 수신시간의 차이로 지연시간을 계산한다(라인 2-3). 다음으로 DB 에 저장된 로드 쉐딩 스위치를 확인한다(라인 4). 만약 지연시간이 ε 보다 크고 로드 쉐딩 스위치가 false 라면, 즉 지연시간이 임계치를 초과했지만 로드 쉐딩이 동작하지 않는다면, 스위치를 true 로 변경해 로드 쉐딩을 동작시킨다(라인 5-6). 반면, 지연시간이 ε 보다 작고 로드 쉐딩 스위치가 true 라면 스위치를 false 로 바꿔 로드 쉐딩을 중지한다(라인 7-8).

```
Input: ε, threshold of processing latency time
Input: socket, socket connected to Kafka consumer
1.  while socket is connected do
2.      Receive send/receive time from Kafka consumer;
3.      latency = receive time – send time;
4.      switch ← Load shedding switch stored in DB;
5.      if latency > ε and switch == false then
6.          Change load_shedding_switch to true;
7.      else if latency < ε and switch == true then
8.          Change load_shedding_switch to false;
```

그림 4. 로드 쉐딩 판별 알고리즘.

그림 5 는 로드 쉐딩 엔진에서 카프카 프로듀서의 메시지 송신 여부 알고리즘이다. 프로듀서는 사용자의 입력 데이터 소스 *source*, 실시간 메시지 처리 시스템의 입력 *input*, 로드 쉐딩 비율 *ratio* 를 입력받는다. 먼저, *source* 에서 메시지가 들어오면 데이터베이스에서 로드 쉐딩 스위치 *switch* 를 확인한다(라인 2-3). 만약 *switch* 가 false 라면, 즉 로드 쉐딩이 동작하지 않는다면 메시지를 실시간 처리 시스템의 입력 *input* 으로 보낸다(라인 4-5). 하지만 *switch* 가 true 라면 로드 쉐딩이 동작 중이므로 로드 쉐딩 비율에 따라 메시지를 버려야 한다. 따라서 0 과 1 사이의 난수를 생성하고, 난수가 주어진 *ratio* 보다 작은 경우에만 메시지를 *input* 으로 보낸다(라인 6-9).

```
Input: source, Input data source
Input: input, Input of the messaging processing system
Input: ratio, Load shedding ratio
1.  while source != EOF do
2.      message ← Receive a message from source;
3.      switch ← Load shedding switch stored in DB;
4.      if switch == false then
5.          send a message to input;
6.      else then
7.          rand ← Random(0, 1);
8.          if rand < ratio then
9.              send a message to input;
```

그림 5. 프로듀서의 메시지 송신 여부 알고리즘.

4. 실험 평가

본 절에서는 제안하는 카프카 로드 쉐딩 엔진의 적합성을 실험으로 평가한다. 실험에 사용한 실시간 처리 응용은 데이터 스트림의 발생지에서 임의로 생성된 실수 배열이 데이터 스트림으로 끊임없이 입력되면, 실시간 처리 시스템이 카프카로부터 배열들을 읽고 정렬하여 다시 카프카로 전송한다. 본 실험에 사용된 실시간 처리 시스템은 아파치 스톰으로, 배열을 정렬하는 토폴로지를 구성하였다. 실험을 수행한 하드웨어 플랫폼은 한 대의 마스터 노드(Xeon E5-2630V3 2.4GHz, 8Core)와 여덟 대의 슬레이브 노드(Xeon E5-2630V3 2.4GHz, 6 Core)로 구성하였으며, 이들 노드는 각기 32GB RAM, 256GB SSD 를 장착하였다. 로드 쉐딩 엔진은 Intel i7 3.60GHz CPU, 8.0GB RAM 을 장착한 HP 워크스테이션에 독자적으로 구성하였다.

첫 번째 실험은 단일 소스에서 로드 쉐딩의 적용 여부에 따른 실시간 처리 응용의 지연시간으비교한다. 처리 지연시간의 임계치 ε 은 5 초(5,000*ms*), 로드 쉐딩 비율 *ratio* 는 0.01 로 설정하고, 실수 1,000 개로 구성된 배열을 1*ms* 마다 입력하였다. 그림 6 은 로드 쉐딩을 적용한 시스템과 적용하지 않은 시스템의 지연시간을 보여준다. 로드 쉐딩이 동작할 때 메시지는 로드 쉐딩 비율만큼 전송되므로, 전송되지 않은 메시지의 지연시간은 추세선으로 표시하였다. 그림을 보면 로드 쉐딩을 적용하지 않은 시스템의 경우, 처리되지 않은 메시지가 계속 쌓여 지연시간이 지속적으로 증가한다. 이는 카프카에 적재된 메시지가 수십 초 이상 지연됨을 의미하며, 이는 실시간성을 전혀 보장할 수 없음을 의미한다. 반면 로드 쉐딩을 적용한 시스템의 경우, 로드 쉐딩 비율을 0.01 로 설정하여 전체 메시지의 1%만 보내 시스템 부하를 제어한다. 지연시간이 임계치인 5 초를 초과하기 전까지는 지연시간이 지속적으로 증가하지만, 5 초를 초과하면 증가하지 않고 일정시간이 흐른 후 지연시간이 매우 짧아져 다시 증가하는 추세를 보인다. 이는 로드 쉐딩 동작 시, 시스템에 유입되는 메시지 수가 줄어 처리 되지 않고 쌓여있던 메시지가 모두 처리되기 때문이다. 즉, 쌓여있던 모든 메시지가 처리된 후, 새롭게 유입된 데이터가 처리되면 지연시간이 매우 짧아지고 이 때 로드 쉐딩이 중지되어 다시 지연시간이 증가하는 것이다.

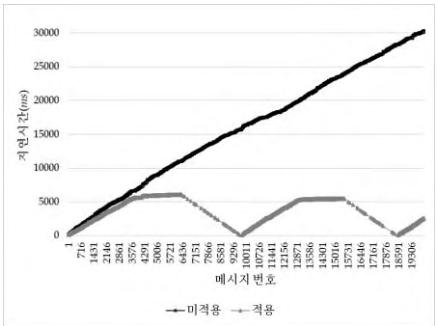

그림 6. 로드 쉐딩 적용 여부에 따른 실시간 처리 엔진의 지연시간 비교.

그림 7 은 로드 쉐딩 엔진을 적용한 시스템에서 입력 소스가 여러 개일 때 각 소스 별 지연시간을 나타낸 것이다. 이전 실험과 마찬가지로 처리 지연시간의 임계치 ε 는 5 초(5,000*ms*), 로드 쉐딩 비율 *ratio* 는 0.01 로 설정하였다. Src1 은 실수 700 개로 구성된 배열, Src2 는 실수 1,000 개로 구성된 배열, Src3 은 실수 1,500 개로 구성된 배열을 1*ms* 마다 입력하도록 설정하였다. 그림을 보면, 데이터 길이에 따른 지연시간의 차이가 있지만 모든 소스의 지연시간이 지속적으로 증가하지 않고 오르내림이 반복되는 추세를 보인다. 따라서, 로드 쉐딩 엔진이 단일/다중 소스일 때 모두 잘 동작하는 것을 확인할 수 있다.

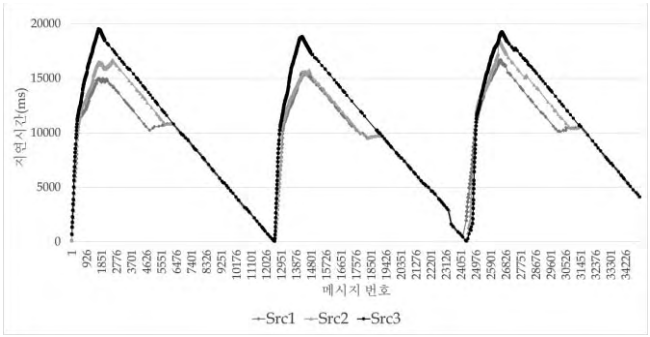

그림 7. ε=5,000, *ratio*=0.01 일 때 다중 소스 실험 결과.

5. 결론 및 향후 연구

본 논문에서는 데이터 스트림의 실시간 처리를 위해 카프카 기반 로드 쉐딩 엔진을 제안하였다. 메시지 처리 지연시간의 지속적 증가를 막기 위해 지연시간이 임계치를 초과할 때 로드 쉐딩이 동작하고, 메시지 버림을 빨리 수행하고 다양한 메시지 처리 플랫폼을 지원하기 위해 카프카 프로듀서에서 로드 쉐딩을 수행하도록 설계하였다. 또한, 시스템의 원활한 동작을 위해 데이터의 특성과 추세를 잘 아는 관리자가 데이터를 얼마나 버릴지 설정하도록 하였다. 실험에서는 실시간 처리 응용으로 아파치 스톰을 사용하여 단일/다중 소스 모두 로드 쉐딩이 효율적으로 작동하는 것을 확인하였다. 본 연구는 데이터 스트림의 입출력을 카프카로 관리하는 실시간 데이터 스트림 처리 시스템에 로드 쉐딩 기법을 적용한 첫 번째 시도로서, 앞으로 스트림 처리에 사용될 의미 있는 연구라 사료된다. 향후 연구로는 비전문가를 위해 로드 쉐딩 비율을 스스로 조정하는 로드 쉐딩 메커니즘을 연구하고 평가한다.

참고문헌

[1] J. Leskovec, et al. A. Rajaraman, and J. D. Ullman, Mining of Massive Datasets, Cambridge University Press, 2014.

[2] B. Babcock, et al., "Models and Issues in Data Stream Systems," In *Proc. of the 21st ACM Symp. on Principles of Database Systems*, Madison, Wisconsin, pp. 1-16, June 2002.

[3] Apache Kafka, http://kafka.apache.org/.

[4] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A Distributed Messaging System for Log Processing," In *Proc. of the NetDB*, Athens, Greece, pp. 1-7, June 2011.

[5] 고승범 and 공용준, 카프카, 데이터 플랫폼의 최강자, 책만, 2018.

[6] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, "Load Shedding in a Data Streams," In *Proc. of the 29th Int'l Conf. on Very Large Data Bases*, Berlin, Germany, pp. 309-320, Sept. 2013.

[7] B. Babcock, et al., "Load Shedding for Aggregation Queries over Data Streams," In *Proc. of the 20th Int'l Conf. on Data Engineering*, Boston, MA, pp. 350-361, Apr. 2004.

[8] Apache Storm, http://storm.apache.org/.

[9] P. Goetz and B. O'Neill, "Storm Blueprints: Patterns for Distributed Real-time Computation," Packt Publishing, Mar. 2014.

[10] Apache Kafka Cluster Architecture, http://www.tutorialspoint.com/apache_kafka/.

[11] Apache Spark Streaming, http://spark.apache.org/streaming/.

[12] Apache Kakfa Streams, http://kafka.apache.org/documentation/streams/.