J. Internet Comput. Serv.
ISSN 1598-0170 (Print) / ISSN 2287-1136 (Online)
http://www.jics.or.kr

안정적인 API 게이트웨이를 위한 스트림 기반 API 조합

Stream-based API composition for stable API Gateway

조 동 일1*
Dong-il Cho

요 약

API 게이트웨이에서 API 조합은 클라이언트의 호출 횟수를 줄이고 오버페칭과 언더페칭을 방지할 수 있는 필수적인 기능이다. IMJ(In-Memory Join)로 동작하는 API 조합은 많은 자원을 소모하여 API 게이트웨이의 성능에 부담을 준다. 본 연구에서는 IMJ 방식의 API 조합의 문제를 개선하기 위해 조합할 데이터를 스트리밍으로 클라이언트에 전달하는 SAPIC(Stream-based API Composition)를 제안한다. SAPIC는 클라이언트 응답 메시지를 구성하는 각각의 MSA API를 호출하여 받은 응답 메시지를 즉시 클라이언트로 스트리밍하여 IMJ에 비해 API 게이트웨이의 자원 소모를 줄이고 빠른 응답시간을 제공할 수 있다. 대표적인 API 조합 기술인 GraphQL 비교 실험결과 SAPIC는 GraphQL에 비해 약 21 ~ 70 % 낮은 최대 CPU 점유율과 약 16 ~ 74 % 낮은 최대 Heap 사용량 그리고 1 ~ 2.3 배의 높은 처리량을 기록하였다.

☞ 주제어 : API Composition, API Gateway, Microservice Architecture

ABSTRACT

In the API gateway, API composition is an essential function that can reduce the number of client calls and prevent over-fetching and under-fetching. API composition that operate with IMJ (In-Memory Join) consume a lot of resources, putting a burden on the performance of the API gateway. In this paper, to improve the problem of IMJ-style API composition, we propose SAPIC (Stream-based API Composition), which delivers the data to be composed to the client by streaming. SAPIC calls each MSA API that makes up the client response data and immediately streams the received response data to the client, reducing the resource consumption of the API gateway and providing faster response time compared to IMJ. As a result of a comparison experiment with GraphQL, a representative API combination technology, SAPIC recorded a maximum CPU occupancy rate of approximately 21 to 70 % lower, a maximum heap usage rate of approximately 16 to 74 % lower, and a throughput rate that was 1 to 2.3 times higher than GraphQL.

☞ keyword : API Composition, API Gateway, Microservice Architecture

1. 서 론

MSA(Microservice Architecture)에서 API 게이트웨이는 외부 API 클라이언트에게 MSA 내부로 접속할 수 있는 진입점을 제공하는 필수 요소이다[1]. API 게이트웨이 사용의 주요 이점은 MSA 애플리케이션의 내부 구조를 캡슐화한다는 것이다[2]. API 게이트웨이는 각 클라이언트에 독립적인 API를 제공하여 클라이언트와 애플리케이션 간의 호출 횟수를 줄이고 클라이언트 코드를 단순화시킬 수 있는 반면 고가용성 구성요소로서 병목 현상을 발생시킬 위험이 크고 서비스의 API를 노출하려면 API 게이트웨이를 변경해야 하는 단점이 있다[3]. 이런 단점에도 불구하고 대부분의 실제 애플리케이션에서는 API 게이트웨이를 사용하는 것이 합리적이다[1,4].

API 게이트웨이는 요청라우팅, API 조합, 프로토콜 변환 그리고 인증 및 권한관리와 같은 엣지기능을 제공할 수 있다[1]. 이중 API 조합은 여러 서비스 API를 호출하고 그 결과를 조합하여 클라이언트에 전송한다[5]. API 게이트웨이는 API 조합을 통해 클라이언트와 애플리케이션 간의 호출 횟수를 줄이고 다양한 클라이언트 기기의 환경에 적합하게 데이터를 가공하여 전달할 수 있다.

현재까지 API 게이트웨이의 API 조합은 IMJ(In-Memory Join) 방식으로 실체화되고 있다[1,5]. IMJ는 API 조합을 위한 구현체를 API 게이트웨이에 설치해야 하고 전체 데이터를 메모리에 로드해야 하며 모든 API 호출이 종료된 후에야 클라이언트로 전송이 가능하기 때문에 배포가 어렵고 서버 자원의 사용률이 높으며 전체 응답속도를 느리

1 Division of R&D, TOMATOSYSTEM, Seoul, 06104, Korea.
* Corresponding author (chodongil@yahoo.co.kr)
[Received 19 October 2023, Reviewed 30 October 2023, Accepted 28 November 2023]

게 한다. 또한 IMJ는 서버측 엔드포인트를 두꺼운 레이어가 되게 하고 이는 쉽게 재앙으로 이어질 수 있으며 결국 별도의 팀에서 관리하게 되고 변경이 발생될 때 마다 변경해야 하는 또 다른 장소가 된다[6].

본 연구에서는 API 조합의 이와 같은 문제를 해결하기 위해 SAPIC(Stream-based API Composition: 스트림 기반 API 조합)를 제안한다. SAPIC는 IMJ와 달리 각 API의 데이터를 스트림으로 처리하고 조합한다. 본 연구에서는 SAPIC를 구성하는 컴포넌트를 설계 및 구현하였고 실험을 통해 널리 사용되고 있는 대표적인 IMJ 방식의 GraphQL과 비교 평가하였다.

2. 관련 연구

2.1 API 게이트웨이

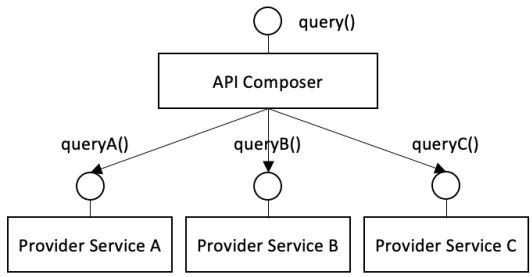
MSA는 애플리케이션을 각각 자체 프로세스에서 실행되고 경량 메커니즘과 통신하는 독립적인 서비스 집합으로 개발하는 방법이다[2]. 클라이언트는 이론적으로 각 MSA에 직접 요청을 보낼 수 있다. 이런 접속 방식은 클라이언트가 접속해야 하는 모든 MSA의 주소를 알고 있어야 하기 때문에 구현과 배포가 어렵고 MSA를 캡슐화 할 수 없게 한다[1].

API 게이트웨이는 모든 클라이언트에 대해 MSA 애플리케이션에 접속할 수 있는 단일 진입점을 제공하고 요청라우팅, API 조합, 프로토콜 변환 그리고 인증 및 권한 처리와 같은 엣지 기능을 제공할 수 있다. MSA에서 API 클라이언트는 API 게이트웨이를 이용하여 호출횟수를 줄이고 클라이언트 특화 프로토콜을 유지하면서 MSA API와 통신할 수 있다[1].

2.2 API 조합

API 게이트웨이의 서비스 중 API 조합은 여러 MSA API에 분산된 데이터를 조합하여 클라이언트에 전송하는 기능으로 클라이언트 별 맞춤형 API를 제공할 수 있어 결과적으로 클라이언트의 요청 수를 줄일 수 있는 API 게이트웨이의 핵심 기능이다[4,7,8].

그림 1과 같이 클라이언트는 API Composer에 원하는 서비스를 요청한다. 일반적으로 API 게이트웨이가 API Composer의 역할을 한다. API Composer는 클라이언트 요청에 응답하기 위해 응답 데이터를 구성하는 Service A, Service B, Service C를 각각 호출하여 그 결과를 조합하여 클라이언트에 전송한다. 대표적인 API 조합 기술로 GraphQL을 이용할 수 있다.



(그림 1) API 조합[5]
(Figure 1) API Composition[5]

2.3 GraphQL

서비스 기반 소프트웨어 아키텍처를 구현하기 위한 새로운 쿼리 언어인 GraphQL은 편리하고 유연하게 서버에서 클라이언트로 데이터를 전달할 수 있다[9,10]. GraphQL은 일반적인 데이터 검색 시 오버페칭 및 언더페칭을 피할 수 있고, 클라이언트가 특정 데이터를 선택할 수 있게 하는 쿼리는 전체 패킷의 크기를 줄여 더 빠른 응답이 가능하며, 클라이언트의 작업도 단순화 할 수 있다. 클라이언트-서버 간의 데이터 통신 링크로 사용되는 GraphQL은 MSA에서 다양한 서비스 API를 관리할 수 있으며 여러 클라이언트에서 사용할 수 있으므로 MSA에서 API 게이트웨이로 사용하기에 적합하다[9].

GraphQL에서는 Schema 정의를 통해 MSA 애플리케이션에서 서비스하는 API의 집합을 사전 정의한다[11,12]. 클라이언트는 Schema를 질의하는 쿼리를 GraphQL 문법으로 작성하여 전송하면 GraphQL 서버는 쿼리의 결과 데이터를 구성할 각 요소 데이터를 조회하는 Fetcher를 실행한 후 결과를 질의문의 구조에 맞게 IMJ 방식으로 조합하여 클라이언트로 전송한다.

이런 GraphQL의 Fetcher는 구현과 배포를 포함해야하기 때문에 별도의 팀에서 관리해야 하고 변경이 발생될 때 마다 로직을 변경해야 하는 또 다른 장소가 된다[6]. 또한 IMJ 방식은 응답속도를 느리게 하며 서버의 자원 소모를 높이는 원인이 된다.

3. Stream-based API Composition

3.1 In-Memory Join API 조합의 문제점

게이트웨이 계층은 모든 요청에 대한 단일 진입점이기 때문에 올바르게 설계하지 않으면 병목 현상이 발생할 수 있다[13,14].

IMJ는 API조합을 위한 구현체 배포가 필요하고 API 조합에 필요한 모든 API의 응답 데이터를 수신한 후에 조합할 수 있기 때문에 다음과 같은 문제점을 가지고 있다.

- API의 배포가 어렵다
- API 게이트웨이의 CPU와 메모리 사용량을 높인다
- 클라이언트 요청에 대한 응답속도가 느리다

본 연구에서는 IMJ의 위와 같은 문제를 해결하기 위해 SAPIC를 제안하고 IMJ와 비교평가를 통해 개선점과 한계를 도출한다.

3.2 API 서비스 모델

최근 웹 데이터는 JSON이나 XML과 같은 편리한 구조화된 형식으로 데이터를 노출하는 REST API를 통해 교환되고, 대부분의 경우 이러한 API의 구조화된 응답은 REST 스타일에 따라 일반적인 구문 형식을 따른다[15]. SAPIC는 클라이언트의 요청이 이런 구조화된 데이터 응답을 소모한다고 가정하고 대표적인 경량 데이터 교환형식인 JSON을 사용한다. JSON은 REST API에서 많이 사용되는 대표적인 데이터 교환형식으로 GraphQL에서도 주요 데이터 형식으로 지원한다.

JSON은 데이터의 구조를 부모/자식/형제 관계로 표현할 수 있다. 부모/자식 관계는 포함관계로 표현되고 형제 관계는 동등한 수준의 데이터를 표현하며 순서에 제약이 없다. 순서 제약이 없이 형제 관계 표현이 가능한 JSON의 특징은 스트림 처리를 유리하게 한다.

일반적으로 API 조합은 다음과 같은 두 종류의 API 간 종속성을 가질 수 있다.

$$C(P1) = A(P1) \cup B(A(P1)) \quad (1)$$

$$C(P1) = A(P1) \cup B(P1) \quad (2)$$

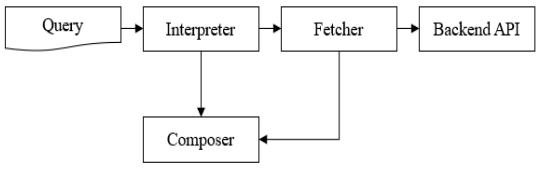
*(1)*에서 조합 서비스 *C*는 서비스 *A*를 호출한 결과 값을 이용하여 서비스 *B*를 호출하고 두 결과를 조합하여 최종 데이터를 생성한다. 이 모델은 부모/자식 관계로 표현될 수 있고 서비스 *C*는 서비스 *A*의 결과를 수집한 후에 서비스 *B*를 호출할 수 있기 때문에 서비스 *A*와 서비스 *B*는 반드시 순차적으로 호출되어야 한다. 이 모델은 호출 순서의 제약사항으로 인해 스트리밍 처리를 할 수 없다. 하지만 Aggregate 패턴을 적용하여 이런 제약사항을 회피할 수 있다[1]. Aggregate 패턴은 마이크로서비스의 경계를 정의할 수 있는 도메인 모델 설계 기법으로 *(1)*과 같은 경우 서비스 *A*와 서비스 *B*는 하나의 조건에 의해 종속된 데이터로 한 API를 통해 서비스가 가능하다. 즉 $D(P1) = A(P1) \cup B(A(P1))$ 로 정의된 서비스*D*를 MSA API로 제공하고 API 게이트웨이에 노출할 수 있다.

*(2)*에서 서비스 *A*와 서비스 *B*는 조건 *P1*에 의해 독립적인 서비스로 조합 서비스 *C*는 서비스 *A*와 *B*의 합으로 표현된다. 이 경우 서비스 *A*와 서비스 *B*의 호출은 병렬로 처리가 가능하고 결과는 조합 서비스 *C*에서 형제관계로 표현할 수 있으며 SAPIC로 처리가 가능하다.

본 연구에서는 *(2)* 서비스 모델을 대상으로 응답시간 및 API 게이트웨이의 자원소모를 줄이는 것을 목표로 한다.

3.3 아키텍처 개요

SAPIC는 그림 2와 같이 Query와 Query를 해석하는 Interpreter, API를 호출하여 데이터를 조회하는 Fetcher 그리고 각 Fetcher에서 병렬로 수집한 MSA API의 응답을 스트림 기반으로 조합하여 클라이언트로 전송하는 Composer로 구성된다.



(그림 2) SAPIC Components
(Figure 2) SAPIC Components

3.4 Query and Interpreter

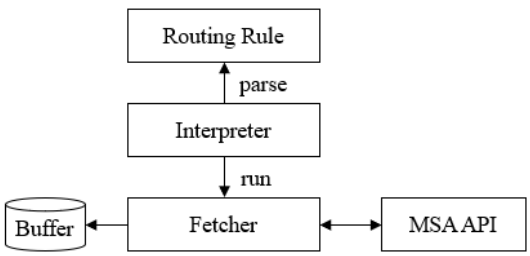
Query는 GraphQL의 Query 문법을 사용하나 GraphQL과 달리 별도의 Schema 정의를 필요로 하지 않는다. Query의 각 노드는 MSA 서비스의 결과데이터 요소거나 상수일 수 있다.

```
routes:
  - predicates:
    - path=/sample1
    client-response:
      content-type: application/json
      body-graph: "query sample1 {
  data {
    influencers: influencers(name: $name) { ...}
    comparison {...}
    comCodes: comCode(div: $div) {...}
    }
  }
}"
    service-targets:
      - name: influencers
        endpoint: ...
      - name: comCode
        endpoint: ...
      - name: comparison
        endpoint: ...
```

(그림 3) 라우팅 규칙 정의
(Figure 3) Routing Rule Definition

Query는 그림 3과 같이 API 게이트웨이에 라우팅 규칙의 일부분으로 정의된다. 라우팅 규칙은 API 게이트웨이에 접속한 요청을 어떤 MSA API로 라우팅할지를 정의한다. API 게이트웨이는 클라이언트의 요청이 *predicates*에 정의된 조건을 만족할 때 *body-graph*에 정의된 Query의 결과를 클라이언트에게 전송한다.

Interpreter는 Query를 해석하여 Query의 각 요소가 *service-targets*의 *name* 필드와 일치할 경우 *service-target*의 API를 호출하는 Fetcher를 자동으로 구성하여 응답데이터의 노드와 맵핑하여 Composer가 조합할 수 있도록 준비한다. *service-targets*의 *name* 필드와 일치하지 않는 요소는 상수 또는 요소를 감싸는 구성 요소로 취급하며 스트리밍 과정에서 즉시 클라이언트로 전송될 수 있다.



(그림 4) Fetcher
(Figure 4) Fetcher

3.5 Fetcher

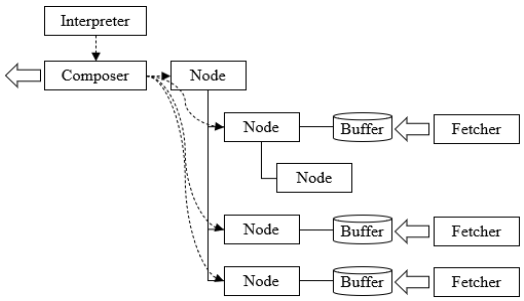
Fetcher는 그림 4와 같이 그림 3에서 정의된 설정을 바탕으로 Interpreter에 의해 자동으로 구성되고, 병렬로 실행되며, 결과를 스트림 방식으로 버퍼에 쌓고 모든 응답 데이터의 수신이 완료되면 버퍼를 닫아 데이터 수신이 종료되었음을 설정한다.

SAPIC는 Schema를 별도로 구성하지 않고 클라이언트가 요구하는 데이터와 그 데이터를 서비스하는 MSA API 호출 정보를 설정을 통해 공급받는다. 즉 Fetcher를 별도로 구현하여 배포하고 실행하는 GraphQL과 달리 SAPIC는 MSA API의 접속 정보를 설정을 통해 정의하고 정의된 정보로 Fetcher를 자동 구성한다. 이런 구성 방식은 Fetcher 배포를 위해 API 게이트웨이를 중단할 필요 없이 데이터베이스와 같은 저장소를 통해 동적으로 설정을 주입하거나 변경할 수 있기 때문에 보다 유연한 유지관리를 가능하게 한다.

또한 MSA API 호출의 결과 데이터는 GraphQL과 달리 엄격한 Schema 정의 없이 각 요소와 MSA API 매핑으로 정의되어 API 게이트웨이의 핵심 서비스와 확장 기능인 MSA API와의 직접적인 결합도를 낮춘다. 게이트웨이 계층은 서비스 별로 여러 인터페이스를 구현해야 하므로 복잡성이 증가한다[13,16]. 핵심서비스와 확장 기능의 결합도를 줄이는 방식으로 최적화된 게이트웨이는 플러그인 모드에 적응하고 사용자 경험을 개선하며 개발 비용을 절감할 수 있다[17].

3.6 Composer

그림 5와 같이 Composer는 Interpreter에 의해 컴파일된 Query를 조합하여 클라이언트에 전송한다. Query는 MSA API의 호출결과가 바인딩되는 부분과 상수 값으로 치환



(그림 5) Composer
(Figure 5) Composer

될 부분으로 나뉜다. Composer는 계층형 데이터 구조인 클라이언트 응답데이터 노드를 순회하면서 클라이언트로 전송할 데이터가 있을 경우 우선적으로 전송한다.

JSON과 같은 경량 데이터 형식에서 객체형의 자식노드의 순서는 문법에 영향을 주지 않는다. 이 특징은 SAPIC에서 동일 깊이의 형제 노드가 둘 이상이고 각 노드가 Fetcher와 연결된 경우 먼저 응답한 API의 데이터를 우선적으로 클라이언트로 전송해도 유효한 응답임을 의미한다. 이 특징은 동일 깊이의 데이터 간에만 유효하고 그 하위 깊이로 진입했을 경우에는 현재 깊이를 기준으로 이 규칙이 적용된다. Composer는 최상위 깊이의 형제 노드들 부터 이 규칙을 적용하여 먼저 전송할 노드를 선택하고 스트리밍한다.

```
Input: Response Message Structure doc

while doc.children.length 〉 0
  for node : doc.children
    if node.buffered == true
      then
        while node.done == false

write(filter(encode(read(node.buffer))));
          compact(node.buffer);
        end while;
        doc.children.remove(node);
    end if;
  end for;
end while;
```

(그림 6) 스트리밍 알고리즘

(Figure 6) Streaming Algorithm

Composer는 Fetcher와 직접연동하지 않고 Fetcher가 MSA API 응답을 버퍼링한 버퍼를 통해 데이터에 접근한다. 그림 6에서와 같이 Composer는 동일 수준의 각 노드가 버퍼링된 데이터가 있는지를 체크*(node.bufferd)*하여 버퍼링된 데이터가 있는 노드를 우선적으로 선택한다. 노드가 선택되면 Composer는 버퍼의 데이터를 읽어와서 *(read)* 응답형식에 맞게 인코딩*(encode)*과 변환*(filter)*과정을 거쳐 클라이언트로 전송*(write)*한다. 버퍼의 데이터는 Composer에 의해 읽히고 나면 메모리에서 제거되어 메모리 사용량을 조절한다*(compact)*. 쓰기 버퍼가 닫힌 버퍼의 모든 데이터가 전송된 후*(node.done)*에는 Composer는 버퍼가 매핑되었던 노드의 형제 노드 중 다음 전송할 노드를 선택하고 동일한 절차를 반복한다. 이와 같이 Query의 모든 노드에 대한 전송이 완료되면 Composer는 자원을 해제한 후 응답을 종료한다.

3.7 SAPIC의 한계

SAPIC는 조합대상인 MSA API를 Fetcher를 통해 병렬로 호출하고 먼저 도착한 데이터를 우선적으로 클라이언트로 스트리밍한다. 이 아키텍처는 모든 MSA API의 실행 이전에 클라이언트 요청이 라우팅되는 즉시 실행되기 때문에 응답은 항상 성공으로 간주되어 전달된다. 반면 GraphQL의 경우 모든 MSA API의 응답을 수신한 후 특정 응답이 실패했을 때 최종 응답을 실패로 처리할 수 있다. 이 차이는 응답데이터를 스트림으로 처리하는 SAPIC의 한계로 이를 극복하기 위해서는 응답데이터의 마지막에 서비스 실행결과를 포함하는 것과 같이 프로토콜 수준의 명세가 필요하다.

4. 구현 및 평가

본 연구에서는 제안한 SAPIC를 구현하고 실험을 통해 GraphQL 기반 API 게이트웨이와 성능을 비교 평가하였다.

4.1 구현 및 실험 환경

GraphQL과 SAPIC는 Java 17 기반에 Spring Boot를 이용하여 구현하였고 구체적인 구현 환경은 표 1과 같다.

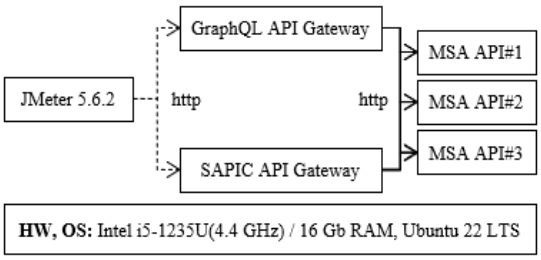
(표 1) 구현환경

(Table 1) Implementation Environment

	GraphQL	SAPIC
Gateway	Spring Cloud Gateway Server 4.0.7 graphql-java 20.2	Implementation
Language	Java 17	
Spring	Spring Boot 3.1.4 Spring WebFlux 6.0.12	

SAPIC와 GraphQL의 공정한 비교평가를 위해 Spring Boot, Spring WebFlux 및 Java 버전은 동일한 버전을 사용했으며 성능에 영향을 줄 수 있는 로그는 OFF로 설정했다. GraphQL에서 API 게이트웨이와 GraphQL 구현체는 Spring Cloud Gateway Server와 graphql-java를 사용하였고 SAPIC는 직접 구현하였다.

비교 평가를 위한 실험환경은 그림 7과 같다.

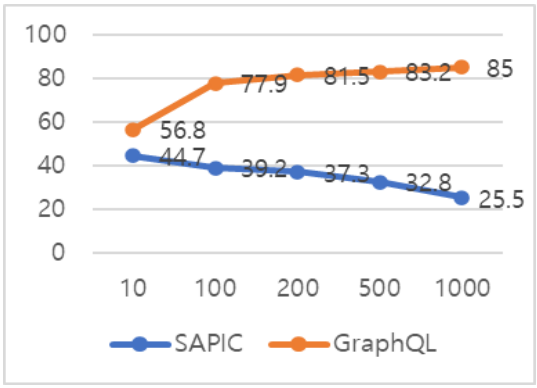


(그림 7) 실험 환경
(Figure 7) Test Environment

JMeter는 클라이언트 부하를 가상화하여 GraphQL과 SAPIC에 각각 HTTP 요청을 전송하고 각 API 게이트웨이는 세 개의 MSA API를 호출한 후 결과를 조합하여 JMeter에 전달한다. GraphQL과 SAPIC는 동일한 MSA API를 호출하며 결과 데이터의 크기는 동일한다. JMeter의 부하 설정은 다음과 같다.

- Number of Threads(users) : 100
- Ramp-up period(seconds) : 5
- Loop Count : 100

테스트케이스는 조합대상 MSA API 각각에서 응답하는 데이터의 건수를 10, 100, 200, 500, 1000으로 증가시키며 테스트하였고 매 케이스 마다 초당 처리량과 API 게이트웨이의 최대 CPU 및 Heap 사용량을 측정하였다.

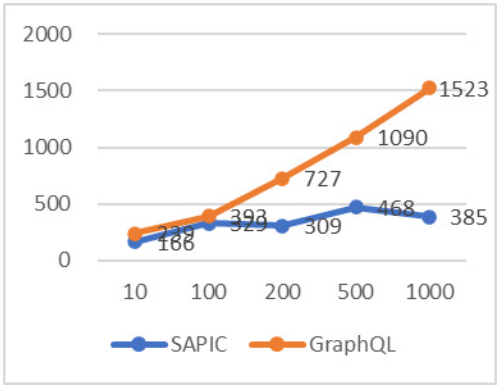


(그림 8) 데이터 건수 별 최대 CPU 사용량(%)
(Figure 8) Max CPU usage(%) by number of data

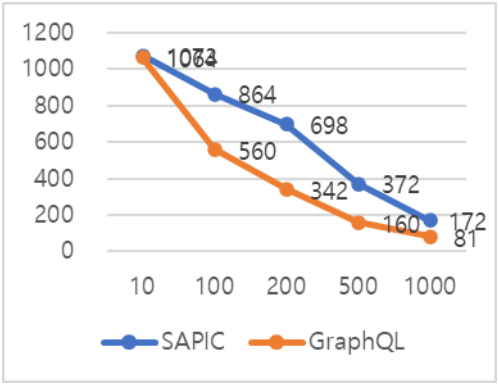
4.2 실험결과 및 분석

그림 8과 같이 CPU 사용량은 데이터 건수의 변화에 따라 SAPIC는 25.5 ~ 44.7 %를 GraphQL은 56.8 ~ 85%를 기록하였다. 데이터 건수 증가에 따라 SAPIC는 CPU 사용량이 감소하는 반면 GraphQL은 증가하는 것을 볼 수 있다.

최대 Heap 사용량은 그림 9와 같이 SAPIC는 166 ~ 385 Mb를 GraphQL은 239 ~ 1,523 Mb를 기록하였다. 그래프에서 확인할 수 있듯이 SAPIC는 비교적 안정적인 Heap을 사용하는 반면 GraphQL은 데이터 건수가 증가함에 따라 Heap 사용량이 선형으로 증가하는 것을 확인할 수 있다.



(그림 9) 데이터 건수 별 최대 Heap 사용량(Mb)
(Figure 9) Max heap usage(Mb) by number of data



(그림 10) 데이터 건수 별 초당 처리량
(Figure 10) Throughput per second by number of data

그림 10에서 보여주는 초당 처리량은 SAPIC는 172 ~ 1073 건, GraphQL은 81 ~ 1064 건으로 SAPIC가 평균적으

로 약 80 % 높은 처리량을 기록하였다.

SAPIC는 데이터 건수에 관계없이 비교적 균일한 CPU와 Heap 사용량을 보이는 반면 GraphQL은 CPU 및 Heap의 사용량이 증가하였다. 특히 데이터 건수가 증가할 때 GraphQL의 Heap 사용량이 선형으로 증가하는 현상은 모든 클라이언트의 요청의 진입점이 되는 API 게이트웨이에서 성능 저하의 원인이 될 수 있다. 처리량 비교에서 GraphQL과 SAPIC 모두 선형적인 감소를 나타냈으나 SAPIC는 보다 완만한 기울기를 보였다

5. 결 론

API 게이트웨이는 MSA에 외부 API 클라이언트의 진입점에 해당하는 서비스를 구현한다. 다수의 서비스로 분리된 MSA의 특성으로 인해 API 게이트웨이의 API 조합은 클라이언트의 호출횟수를 줄이고 오버페칭과 언더페칭을 방지할 수 있는 필수적인 기능인데 현재까지 IMJ로 동작하는 API 조합은 높은 자원을 소모하여 API 게이트웨이의 성능에 큰 영향을 준다.

본 연구에서는 IMJ API 조합의 문제점을 개선하기 위해 조합할 데이터를 스트리밍으로 클라이언트에 전달하는 SAPIC를 제안하였다. SAPIC는 클라이언트 응답 메시지를 해석한 후 응답데이터를 구성하는 각각의 MSA API를 호출하고 즉시 응답메시지를 클라이언트로 스트리밍한다. 데이터의 계층적 표현방식인 JSON과 같은 메시지 형식은 같은 계층의 형제 노드의 순서는 메시지 형식에 영향을 주지 않으므로 동일계층의 노드 중 먼저 버퍼링된 노드부터 클라이언트로 전송함으로써 API 게이트웨이의 CPU 및 Memory 사용량을 줄이면서 빠른 응답이 가능하다.

본 연구에서 SAPIC의 검증을 위해 실시한 GraphQL과의 비교실험에서 SAPIC는 GraphQL에 비해 약 21 ~ 70 % 낮은 최대 CPU 점유율과 약 16 ~ 74 % 낮은 최대 Heap 사용량 그리고 1 ~ 2.3 배의 높은 처리량을 기록하였다. 데이터양의 변화에 따른 본 테스트에서 SAPIC는 데이터 변화에도 비교적 균일한 CPU 및 Heap 사용량을 보인 반면 GraphQL은 특히 Heap 사용량에서 선형증가를 보여 API 게이트웨이의 성능을 저하시킬 수 있는 원인이 될 수 있었다.

반면 SAPIC는 스트림 처리를 위한 한계점이 존재한다. IMJ는 모든 MSA API의 응답을 수신한 후 조합하기 때문에 정책에 따라 부분실패 및 전체실패에 대한 처리가 쉽게 가능하지만 SAPIC는 MSA API의 최종 응답결과와 관계 없이 무조건 성공한 응답의 전송만 가능하다. 이 문제는 서비스 호출의 성공/실패 여부를 응답데이터의 마지막에 포함하는 것과 같은 프로토콜 수준의 지원을 필요로 한다.

참고문헌(Reference)

[1] C Richardson, “Microservices Patterns,” Manning, pp.152-160, 253-291, 2018.

[2] Q Xiong and W Li, “Design and Implementation of Microservices Gateway Based on Spring Cloud Zuul,” 3rd International Conference on Computer Information and Big Data Applications, pp.1-5, 2022. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9899125&isnumber=9898665

[3] X Gao, R Liu and X Lin, “API Gateway Optimization Architecture Based on Heterogeneous Hardware Acceleration,” IEEE 3rd International Conference on Information Technology and Big Data and Artificial Intelligence, pp. 863-868, 2023. https://doi.org/10.1109/ICIBA56860.2023.10165387

[4] P D Francesco, P Lago and I Malavolta, “Architecting with microservices: A systematic mapping study,” Journal of Systems and Software, Vol.150, pp.77-97, 2019. https://doi.org/10.1016/j.jss.2019.01.001

[5] Pattern: API Composition. http://microservices.io/patterns/data/api-composition.html

[6] S Newman, “Building Microservices: Designing Fine-Grained Systems,” pp.44-45, O’Reilly Media, 2021.

[7] F M Imani, “Aggregator Backend API With KrakenD,” Proceeding International Applied Business and Engineering Conference, Vol.2, pp.229-233, 2022. https://abecindonesia.org/iabec/index.php/iabec/article/view/108

[8] N Vohra and I B K Manuaba, “Implementation of REST API vs GraphQL in Microservice Architecture,” International Conference on Information Management and Technology, pp.45-50, 2022. https://doi.org/10.1109/ICIMTech55957.2022.9915098

[9] M F Radhiyan, D Khairani and H B Suseno, “Analysis and Design of Microservices Architecture with GraphlQL as an API Gateway for Higher Education Information System,” International Conference on Science and Technology, pp.1-7, 2022.
https://doi.org/10.1109/ICOSTECH54296.2022.9829090

[10] G Brito and M T Valente, “REST vs GraphQL: A Controlled Experiment,” IEEE International Conference on Software Architecture, pp.81-91, 2020.
https://doi.org/10.1109/ICSA47634.2020.00016

[11] E Porcello, A Banks, “Learning GraphQL - Declarative Data Fetching for Modern Web Apps,” O’Reilly Media, pp2-5, 2018.

[12] O Hartig and J Pérez, “An initial analysis of Facebook’s GraphQL language,” International Workshop on Foundations of Data Management and the Web, pp. 1-10, 2017.
https://ceur-ws.org/Vol-1912/paper11.pdf

[13] D Malavalli and S Sathappan, “Scalable microservice based architecture for enabling DMTF profiles,” International Conference on Network and Service Management, pp.428-432, 2015.
https://doi.org/10.1109/CNSM.2015.7367395

[14] J Lin, L C Lin and S Huang, “Migrating web applications to clouds with microservice architectures,” International Conference on Applied System Innovation, pp.1-4, 2016.
https://doi.org/10.1109/ICASI.2016.7539733

[15] D Serrano, E Stroulia, D Lau and T Ng, “Linked REST APIs: A Middleware for Semantic REST API Integration,” IEEE International Conference on Web Services, pp.138-145, 2017.
https://doi.org/10.1109/ICWS.2017.26

[16] W Scarborough, C Arnold and M Dahan, “Case study: Microservice evolution and software lifecycle of the xsede user portal api,” in Proc. of Conference on Diversity Big Data and Science at Scale, No.47, pp.1-5, 2016.
https://doi.org/10.1145/2949550.2949655

[17] Z Xianyu, S Yuehan, W Qianqian and X Yi, “An API gateway design strategy optimized for persistence and coupling Advances in Engineering Software,” Elsevier, Vol.148, 2020.
https://doi.org/10.1016/j.advengsoft.2020.102878

◐ 저 자 소 개 ◑

조 동 일(Dong-il Cho)
2003년 수원대학교 기계공학과(공학사)
2008년 숭실대학교 정보과학 대학원 소프트웨어공학과(공학석사)
2012년 숭실대학교 대학원 컴퓨터학과(공학박사)
2003년~현재 (주)토마토시스템 기술연구소 연구소장
관심분야 : Microservice Architecture, 웹 애플리케이션 아키텍처, WEB UI/UX, 미들웨어, etc.
E-mail : chodongil@yahoo.co.kr