KIPS Tr. Comp. and Comm. Sys.
Vol.7, No.6 pp.145~154 pISSN: 2287-5891

https://doi.org/10.3745/KTCCS.2018.7.6.145

Active-Active Message Replica Scheme for Enhancing Performance of Distributed Message Broker

Kyeonghee Seo[†] · Sangho Yeo[††] · Sangyoon Oh[†††]

ABSTRACT

A loosely coupled message broker system is a popular method for integrating distributed software components. Especially, a distributed broker structure with multiple brokers with active-standby or active-active message replicas are used to enhance availability as well as message processing performance. However, there are problems in both active-standby and active-active replica structure. The active-standby has relatively low processing performance and The active-active structure requires a high synchronization overhead. In this paper, we propose an active-active structure of replicas to increase the availability of the brokers without compromising its high fault-tolerancy. In the proposed structure, standby replicas process the requests of the active replicas so that load balancing is achieved without additional brokers, while the distributed coordinators are used for the synchronization process to decrease the overhead. We formulated the overhead incurred when synchronizing messages among replicas, and the formulation was used to support the experiment results. From the experiment, we observed that replicas of the active-active structure show better performance than the active stand-by structure with increasing number of users.

Keywords : Availability, Distributed Broker, Load Balancing, Performance, Replica

분산 브로커의 가용성 향상을 위한 메시지 레플리카 액티브-액티브 구조 기법

서 경 희[†] · 여 상 호[††] · 오 상 윤[†††]

요 약

다양한 분산 소프트웨어 컴포넌트들의 정보 교환을 위해 비동기, 다대다 메시지 교환이 가능한 브로커 구조가 보편적으로 사용되고 있다. 특별히 많은 사용자 및 메시지를 지원하기 위해 높은 확장성의 분산 메시지 브로커가 제안되었다. 브로커의 가용성 및 장애 극복 능력을 향상시키기 위해 메시지 레플리카를 사용하여 액티브-스탠바이 혹은 액티브-액티브 구조를 사용하게 된다. 그러나, 액티브-스탠바이의 경우 낮은 가용성의 문제, 그리고 액티브-액티브의 경우 동기화 오버헤드가 전체 성능을 낮추는 문제를 가진다. 본 논문에서는 장애 상황의 극복이 가능하면서도 분산 메시지 브로커의 가용성을 향상시키기 위해 메시지 레플리카를 액티브-액티브 구조로 구성하여 분산 브로커의 요청 부하를 분산시키는 기법을 제안하였다. 스탠바이 레플리카들이 액티브 레플리카로부터 요청을 전달받아 나누어 처리함으로써 브로커를 구성하는 노드 수의 증가 없이 요청 부하를 분산시킬 수 있었다. 이때 메시지 동기화 과정은 분산 코디네이터를 이용, 분산 락을 구현함으로써 모든 액티브 레플리카들이 한 때에 동기화를 진행하도록 하였고 각 액티브 레플리가 동기화를 할 때보다 추가적인 오버헤드를 적게 하였다. 본 제안의 성능을 평가하기 위해 제안 기법과 기존의 액티브-스탠바이 기법을 기반으로 브로커 프로토타입을 구현하고 메시지의 생산, 소비 및 전체 생산-소비 구간 처리 성능을 측정 비교하였고, 분산 락으로 인한 오버헤드 수식을 제시하였다. 실험 결과에서 본 제안 기법이 더 높은 확장성과 메시지 처리성능을 보임을 확인하였다.

키워드 : 가용성, 분산 브로커, 부하 분산, 처리 성능, 레플리카

1. 서 론

네트워크 상에 분산되어 있는 소프트웨어 컴포넌트들 간의 정보교환을 위해서 다양한 메시징 방식이 제안되어 왔다. 초기에 두 컴포넌트의 동기적 연결을 통한 메시징을 가능하게 하는 원격 프로시저 호출(Remote Procedure Call, RPC)과

※ 이 논문은 2017년도 정부(교육부)재원으로 한국연구재단의 지원을 받아 수행된 연구임(NRF-2015R1D1A1A01059557).
† 비 회 원 : (주) NHN 사원
†† 비 회 원 : 아주대학교 컴퓨터공학과 석사과정
††† 종신회원 : 아주대학교 소프트웨어학과 교수
Manuscript Received : December 15, 2017
First Revision : April 19, 2018
Accepted : April 20, 2018
* Corresponding Author : Sangyoon Oh(syoh@ajou.ac.kr)

같이 시간 및 공간적 동기를 요구하는 방식으로부터, 동기화의 제한점을 극복하기 위해 비동기 방식의 메시지 큐(message queue)나 메시지 브로커(message broker)와 같은 다양한 패러다임이 등장하였다. 이러한 비동기 방식의 메시징에서 사용하는 메시징 패러다임으로 점대점 패러다임(point-to-point)이나 발행/구독 패러다임(publish/subscribe) 등이 사용된다[1-3].

다양한 메시징 방식 중 브로커 기반의 메시징 방식은 다양한 메시징 패러다임을 사용할 수 있어서(예: 점대점 모델과 발행/구독 모델 등) 많은 적용 사례들을 가진다[4]. 초기 브로커 구조에서는 메시징 시스템이 제공해야 하는 핵심 기능들, 즉 guranteed delivery, persistent messaging, 그리고 message sequencing과 같은 기능들을 제공함에 있어서 단순한 구조의 적용이 가능한 단일 브로커 노드 구조가 보편적이었으나, 이후 많은 사용자를 지원하는 것이 가능하도록 메시지 처리의 분산을 위해 다수의 브로커를 결합하여 브로커 연합 (broker federation)을 구성하는 분산 브로커 구조를 사용하는 시스템이 증가하고 있다.

분산 브로커 구조에서는 복수 노드를 사용함으로서 많은 사용자 요청을 처리할 수 있는 높은 성능을 달성할 수 있으며, 단일 브로커 구조가 가지는 브로커의 single-point of failure 문제를 극복할 수 있어 상대적으로 높은 장애 극복 능력(fault-tolerance)을 가진다. 장애 극복 능력 향상을 위한 요소로서 많은 분산 브로커 구조에서는 노드 간 액티브- 스탠바이 구조 채용하고 있다. 이 구조에서는 액티브 노드에서 처리한 사용자의 메시지를 복제하여 레플리카를 만들고, 이 레플리카를 여러 스탠바이 노드에 저장함으로써 장애 극복 능력을 향상시킨다. 레플리카들은 주기적으로 액티브의 변경내용을 업데이트 하여야 하며, 만약 액티브가 장애상황 등 비가용 상태가 되면 액티브에 가장 가까운 스탠바이 레플리카 중 하나가 기존의 액티브를 대체하여 사용자의 요청을 대신 처리하게 된다. 이렇게 액티브-스탠바이 구조를 통해 신속한 장애 극복이 가능하지만, 동시에 장애 극복 성능 및 메시징 시스템의 전체 성능이 액티브와 레플리카 간의 동기화 주기와 밀접한 관계를 가지는 문제를 내포한다. 동기화 주기가 길수록 장애가 발생했을 때 스탠바이 레플리카로 복구할 수 없는 메시지의 수가 증가하며, 동기화 주기가 짧을수록 레플리카들의 동기화 오버헤드로 인해 전체 메시징 시스템의 성능이 낮아지는 문제가 발생한다.

모든 노드가 동일한 데이터를 가지고 액티브로서 사용자 요청을 처리하는 액티브-액티브 구조는 장애 극복 능력과 사용자 요청 처리 성능 면에서 단일 노드 기반의 브로커 구조뿐 아니라 액티브-스탠바이 구조의 제한 및 문제점을 극복하는 구조이다. 다만, 액티브-액티브 구조를 달성하기 위해서는 복잡한 동기화 알고리즘이 반드시 필요하며, 대부분의 구현에서 높은 동기화 오버헤드가 발생하기 때문에 보편적으로 사용할 수 있는 일반적인 분산 브로커 구조에서는 적용되지 못하는 문제가 있다.

본 논문에서는 높은 장애 극복 능력을 가지지만 제한적으로만 확장되어 높은 처리성능을 달성하기 어려운 액티브-스탠바이 구조의 제한점을 극복하기 위해 변형 액티브-액티브 구조를 제안하여 레플리카를 액티브로 활용하는 기법을 제안한다. 본 제안에서는 액티브-액티브 구조를 달성하기위해 필요한 복잡한 동기화 과정을 분산 코디네이터에 의한 분산 락을 구현으로 단순화하였으며, 높은 처리 성능의 달성을 위해서 original 액티브 노드에 로드 밸런서 기능을 추가하여 다수의 레플리카-액티브 노드에 사용자 요청을 분산하도록 설계하였다. 다만, 동기화되지 않은 메시지를 가진 액티브-레플리카가 비가용 상태가 되었을 때 해당 레플리카가 다시 가용 상태가 되기 전까지는 복제되지 않은 메시지를 이용할 수 없는 문제는 향후 추가 연구에서 해결하여야 한다.

본 논문의 이후 구성은 다음과 같다. 2장에서는 분산 시스템에서 Single Point of Failure를 해결하기 위해 이중화 방법을 적용한 연구, 그리고 제안 기법에서 사용한 분산 코디네이터에 대해 간단하게 소개한다. 본 연구에서 제안하는분산 메시지 브로커의 가용성 향상을 위한 변형 액티브-액티브 구조 제안 기법은 3장에서 소개한다. 4장에서는 제안 기법의 효과성을 입증하기 위한 실험 결과를 설명하며, 5장에서는 결론과 향후 연구에 관해 서술한다.

2. 관련 연구

본 장에서는 분산 시스템에서 SPOF를 해결하기 위해 이중화 방법을 적용한 사례들과 본 제안 기법에서 메시지 동기화를 위해 사용한 분산 코디네이터 기법에 대해 설명한다.

2.1 분산 시스템에서 SPOF의 해결

시스템에 존재하는 Single Point of Failure를 제거하는 방안으로 SPOF에 해당하는 요소를 복수화 하는 방법을 주로 사용하며, 크게 액티브-스탠바이와 액티브-액티브 구조가 있다[5].

1) 액티브-스탠바이 구조

아파치 하둡[6]은 맵리듀스 방식의 오픈 소스 분산 병렬처리 프레임워크로, 분산 병렬 프로그래밍의 단순화와 다중 사용자 환경 지원 등의 장점으로 인해 사용자가 증가하고 있다. 하둡의 데이터 저장에는 HDFS(Hadoop Distributed File System)를 사용하는데, HDFS는 데이터노드(datanode)와 네임노드(namenode)로 구성되어 있다. HDFS의 데이터노드는 기본적으로 3개의 복제본을 두어서 SPOF의 문제가 없는 반면 네임노드는 HDFS의 마스터 역할로 하나의 노드로 구성되었기 때문에 SPOF 문제를 가진다[7]. 하둡 버전 2부터는 네임노드의 SPOF 문제를 해결하기 위해서 액티브 네임노드와 스탠바이 네임노드를 두는 이중화 방법을 사용했고, 네임노드를

가용할 수 없는 상황이 되면 스탠바이 네임 노드가 대체한다. 액티브 네임노드와 스탠바이 네임노드 사이에서 동기를 맞추는 방법으로는 Quorum Journal Manager 나 NFS 등을 사용한다. HDFS의 네임노드는 데이터노드 관리를 위한 하트비트나 메타데이터 같은 작은 크기의 메시지를 주기적으로 다루기 때문에 액티브-스탠바이의 구조가 적절하다.

Amnesia [8]은 분산 스트리밍 프로세싱 엔진에서의 SPOF 문제를 액티브-스탠바이 방식의 이중화를 사용하여 해결하기 위해 제안된 차이 복구(gap recovery) 방법으로, 실패 복구 시 런타임 오버헤드가 적은 것이 특징이다. 실패 복구 과정은 액티브 노드를 사용할 수 없을 때 스탠바이 노드가 액티브 노드로 변환할 때 진행되는 과정으로, 제안된 차이 복구 방법은 인풋이나 아웃풋에 대한 보호가 없어 노드 실패를 복구하는 과정에서 데이터 손실이 일어날 수 있다. 그렇지만 이 연구에서는 분산 스트리밍 프로세싱 엔진이 사용되는 환경에서 노드 실패가 발생했을 때 예전 데이터가 약간 손실되더라도 복구의 보장성보다는 복구되는 시간과 런타임 오버헤드가 적은 것을 중요하게 생각하여, 복구 시간과 런타임 오버헤드가 비교적 큰 정밀 회복(precise recovery)이나 롤백 복구(rollback recovery)보다 차이 복구 방법을 사용하였다.

2) 액티브-액티브를 사용한 연구들

웹 서버에서 순간적으로 발생하는 많은 양의 트래픽을 처리하기 위해서, 서버를 여러 개의 액티브 노드로 구성하고 그 앞에 로드 밸런서를 두어 사용자 트래픽을 라운드 로빈하게 각 노드로 분산시키는 방법을 기본적으로 사용한다. 일반적으로 웹 서버가 처리하는 요청들은 이전 요청에 대한 의존성 없이 각각 독립적으로 수행할 수 있기 때문에 요청 처리 노드의 액티브-액티브 구성을 통해 한 번에 많은 양의 요청을 처리할 수 있어 효과적이다. 로드 밸런싱을 위해서 라운드 로빈 외에도 least connections, hash, IP hash 같은 방법을 사용되어 트래픽을 분산시킨다[9]. 이때는 로드 밸런서가 SPOF가 될 수 있기 때문에 장애를 허용할 수 있게 잘 설계된 로드 밸런서를 사용하는 것이 중요하다. 웹 서버 또한 액티브-스탠바이 구조로 구성할 수 있으나 처리하는 요청이 독립적인 웹 서버에서는 하나의 액티브 노드로 요청을 처리하는 것보다 여러 액티브 노드로 요청을 처리하는 것이 더 효과적이다.

2.2 분산 코디네이터

분산 시스템이 높은 신뢰성과 장애 허용을 제공하며 별다른 문제 없이 동작하기 위해서는 시스템을 이루는 여러 노드의 상태 동기와 메타데이터 등에 대해 관리가 필요하다. 아파치 주키퍼[10]은 분산 시스템의 메타데이터 관리 등을 위한 분산 코디네이터로서, 디렉토리 구조를 가지며 키-밸류 형식으로 데이터를 저장한다. 주키퍼에서는 데이터 저장을 위해 영구 노드(persistent node), 임시 노드(ephemeral node), 시퀀셜 노드(sequential node)와 이들의 변화를 감지하는 와처(watcher)를 제공한다. 주키퍼를 사용하여 분산 시스템을 구성하는 메타데이터 등을 저장하고 관리할 수 있으며, 주키퍼에서 제공하는 여러 종류의 노드들과 watcher를 응용하여 분산 시스템 내에서의 글로벌 분산 락이나 큐 등을 구현할 수 있다.

3. 액티브-액티브 구조를 활용한 분산 브로커 성능 향상 기법

분산 메시지 브로커는 단일 서버 기반의 중앙 집중식 메시지 브로커가 가지는 부족한 확장성, 부족한 장애 극복 능력 문제를 극복하기 제안되었다. 이러한 분산 메시지 브로커의 구조로는 카프카에서 사용하는 것과 유사한 레플리카기반의 액티브-스탠바이 구조를 많이 사용한다[11-13]. 그러나, 액티브의 요청 처리 능력의 한계를 액티브 수를 틀리는 scaling을 통해서만 해결이 가능하며, 이를 해결하기 위해 액티브-액티브를 달성하려는 노력들도 많이 진행되어 왔다.

그러나, 액티브-액티브 구조가 액티브-스탠바이 구조 대비 가지는 우수한 장애 극복 성능 및 요청 처리 능력에도 불구하고 액티브-액티브를 달성하기 위해 필요한 동기화 오버헤드 및 동기화 과정의 복잡도로 인해 일부 시스템에서만 제한적인 목적으로 사용되어 왔다. 본 논문에서는 분산 메시지 브로커에 액티브-액티브 구조의 레플리카를 적용하는 기법을 제안한다. 제안하는 기법은 브로커를 구성하는 노드 수의 증가 없이 부하를 분산시키는 것을 목표로 하며, 분산 코디네이터를 이용한 분산 락을 통해 각 액티브 레플리카들의 메시지 동기를 맞추었다. 이때 락 한 번에 모든 레플리카의 동기화를 같이 진행함으로써 각 레플리카가 따로 동기화를 맞추면서 생기게 되는 추가적인 오버헤드를 없앴다.

3.1 액티브-액티브 레플리카 구조 정의

액티브-스탠바이 구조를 기반으로 액티브-액티브를 달성하기 위해서 각 레플리카의 역할을 정의하고 필요한 요소들을 분석한다. 기존 사용자 액티브-스탠바이 구조의 액티브 레플리카를 오리지널 레플리카로 정의하고, 기존 스탠바이 레플리카는 오리지널로부터 사용자 요청을 전달받아 처리하는 세컨더리 레플리카로 정의한다. 오리지널 레플리카는 해당 노드에 로드 밸런서 기능을 추가하여 도착하는 사용자 요청을 세컨더리 레플리카에 분산하여 오리지널 레플리카에서만 처리하던 사용자 요청을 세컨더리에서 처리할 수 있게 한다.

이를 위해서 기존에 액티브 레플리카(즉, 오리지널 레플리카)의 상태 동기만 맞추던 기존의 스탠바이 레플리카(즉, 세컨더리 레플리카)들은 오리지널 레플리카 뿐 아니라 자신과는 독립적으로 사용자의 요청을 처리했을 다른 세컨더리 레플리카들과도 동기를 맞춰야 한다. 제안 기법에서는 액티브-액티브 구조를 적용하면서 생기는 동기화 과정의 추가적인 오버헤드를 줄이고, 레플리카들의 메시지 동기 과정을 코디

네이트 하기 위해 아파치 주키퍼와 같은 분산 코디네이터를 이용해 분산 락을 구현했다. 락이 걸려 있는 동안 모든 레플리카가 같은 순간에 메시지의 동기를 맞추도록 하였다.

동기화 과정은 모든 세컨더리 레플리카는 동일한 주기(예: 수 초)로 레플리카 간 메시지 동기화 과정을 가지며, 동기화가 필요한 한 세컨더리 레플리카가 락을 거는 것으로 시작된다. 또한, 모든 세컨더리 레플리카는 자신의 동기화 과정 요청을 위해 분산 락을 생성할 수 있으며, 분산 락 생성으로 인한 watcher의 알림은 해당 세컨더리 레플리카의 하트비트 역할도 같이 한다. 한 번 동기화 과정이 시작된 이후에는 모든 세컨더리 레플리카가 동기화에 참여하기 때문에 새로운 락이 발생할 가능성능 낮으나, 시간차에 의한 추가 생성 락은 무시됨으로써 기존의 동기화 과정이 다른 락으로 인해 방해받는 경우가 발생되지 않는다. 동기화 주기가 레플리카마다 다르게 지정된 경우 동기화 과정, 즉 락이 생성되고 동기화가 일어난 후 락이 풀리는 일련의 과정이 연속해서 일어날 수 있다. 첫 번째 동기화 대비 두 번째 동기화부터는 동기화가 필요한 메시지의 양은 상대적으로 적어져서 이후의 오버헤드는 주로 분산 락의 생성 오버헤드이다.

사용자의 요청 처리과정은 카프카의 사용자 요청 처리 과정과 유사하다. 카프카에서는 메시지를 생산할 때 어떤 액티브 레플리카에 저장할지에 대한 파티셔닝 방식으로 기본적으로 라운드 로빈을 제공하며, 이때 메시지 생산자가 특정 파티션 값이나 키 값을 이용하지 않는다면 메시지는 모든 파티션에 골고루 분산되어 특정 액티브 레플리카에 요청 부하가 발생하지 않는다. 그러나 메시지 생산자는 메시지를 생산할 때 특정 키 값을 사용할 수 있고 메시지 생산자마다 파티셔닝에 사용한 방법은 사용자의 요구에 따라 각각 다를 수 있다.

만일 카프카에 의해 사용된 메시지 파티셔닝 방법과 본 제안에서 오리지널 레플리카가 요청을 분산할 때 사용하는 방법이 일치한다면 요청의 처리를 분산시키더라도 계속해서 특정 레플리카로 메시지가 전달되어 부하 불균형이 발생할 수 있다. 그러므로 오리지널 레플리카가 다른 세컨더리 레플리카에 요청을 분산할 때에는 모듈러 연산이나 다른 잘 알려진 해시 알고리즘을 이용하여 로드 밸런싱을 하는 것 보다 모든 레플리카에 공평하게 요청을 전달하기 위해서 라운드 로빈 방법을 사용했다. 또한, 레플리카들의 동기화를 위해 필요한 메타데이터의 양을 최소화하기 위해 오리지널 레플리카에서 사용자의 모든 요청을 우선 수용한 후 세컨더리 레플리카에는 생산 요청과 소비 요청만 분산하였다. 따라서 제안 기법에서는 사용자의 생산 요청과 패치 요청만 세컨더리 레플리카에 분산되도록 하고 메타데이터 요청과 같은 다른 요청들은 오리지널 레플리카에서 처리하도록 하였다.

3.2 레플리카 간 메시지 동기화 과정

제안 기법에서 메시지 동기화 과정은 분산 락을 만드는 것을 시작으로 진행되며 어느 한 세컨더리 레플리카에서 메시지 동기화를 위해 락을 생성하면 그 순간 오리지널 레플리카를 포함한 모든 레플리카가 메시지 동기를 맞추는 것을 시작한다. 레플리카들끼리 메시지 동기를 맞추기 전에 먼저 진행되는 과정은 다음과 같다. 먼저 오리지널 레플리카는 시작할 때 코디네이터에 'syncStart'와 'syncEnd', 'lock'노드가 있는지 확인하고, 해당 노드들이 없다면 각각을 생성한다. syncStart와 syncEnd 노드에는 레플리카들이 마지막으로 동기를 맞춘 메시지의 오프셋과 현재 동기를 맞추고자 하는 마지막 메시지 오프셋이 각각 적힌다. lock 노드에는 다음에 동기화를 원하는 레플리카가 자신의 브로커 아이디로 자식 노드를 만든다. lock 노드에 브로커 아이디를 가진 자식 노드를 생성하고 syncEnd 노드 값을 업데이트함으로써 분산 락을 구현하고, 분산 락이 걸렸을 때 모든 레플리카가 동기화 과정을 실행한다. 동기화 과정은 락이 걸려있는 동안 중복되게 진행되지 않도록 한다. 레플리카들이 분산 락이 걸렸음을 알 수 있도록 모든 레플리카는 syncEnd의 변경과 lock의 자식노드 여부 변경에 대한 watcher를 만든다. 레플리카 사이의 메시지 동기화는 세컨더리 레플리카들이 브로커링에 참가하여 코디네이터의 두 노드에 watcher를 만든 이후부터 수 초 간격으로 이루어지며, 각 레플리카의 동기화 주기는 같다.

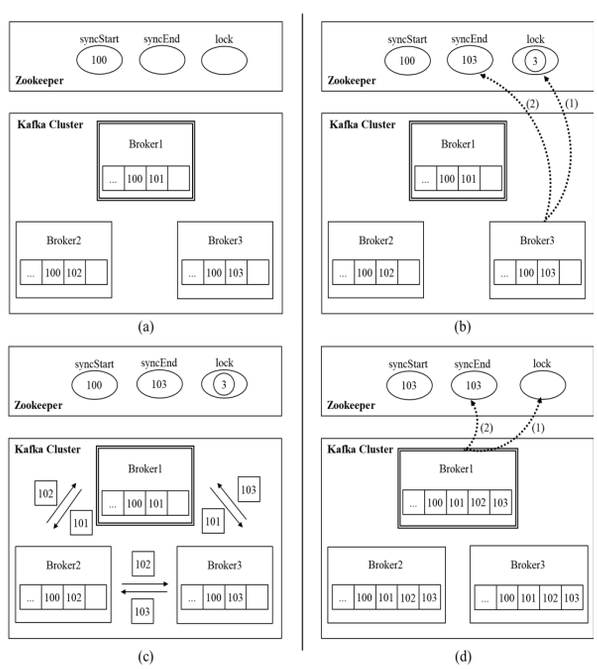

Fig. 1. Message Synchronization Processing

• **메시지 동기를 맞추기 전** (before synchronization)

Fig. 1(a)는 동기를 맞추기 전 각 레플리카와 코디네이터에 있는 노드들의 상태를 나타낸다. syncStart 값을 통해 레플리카들은 오프셋이 100인 메시지까지 동기가 맞추어져 있으며, 각각의 레플리카에는 동기화되지 못한 메시지가 있음을 알 수 있다.

• 메시지 동기 시작 (after synchronization)

Fig. 1(b)는 메시기 동기화 과정의 시작을 나타낸다. 먼저 (1) 세컨더리 레플리카인 Broker3이 메시지 동기를 맞추기 위해 'lock' 노드에 자신의 브로커 아이디를 가진 자식 노드를 만든다. (2) Broker3은 syncEnd의 값을 자신이 가진 마지막 메시지 오프셋 값인 103으로 업데이트함으로써 분산 락이 생성된다. lock 노드에 자식 노드가 생성되면 오리지널 레플리카는 해당 자식 노드 값을 통해 어떤 레플리카가 살아있는지 확인한다. 'lock'노드에 자식 노드가 생기고 'syncEnd'노드 값이 변경되면 락이 걸리며, 등록한 watcher를 통해 모든 레플리카에 분산 락이 생성되었음이 전달된다.

• 동기를 맞출 메시지 전송

분산 락이 생성된 후 알림을 받은 모든 레플리카는 다른 레플리카와 syncStart이후부터 syncEnd까지의 메시지를 서로 복제한다. Fig. 1(c)는 동기화 과정 중 메시지 전송이 일어나는 과정으로, 여기서는 101부터 103 사이의 오프셋을 가진 메시지를 주고받는다. 이때, Broker1은 오프셋 101의 메시지를 Broker2와 Broker3에게 전송하고, Broker2는 오프셋 102의 메시지를 Broker1과 Broker3에게 전송하며, Broker3은 오프셋 103의 메시지를 Broker1과 Broker2에게 전송한다. 다른 레플리카들로부터 메시지를 다 받은 레플리카는 메시지를 디스크에 저장하고 동기화를 맞췄다는 요청을 오리지널 레플리카에 보낸다.

• 메시지 동기 완료

모든 세컨더리 레플리카들은 메시지 동기화를 마치고 나면 오리지널 레플리카에게 동기화를 완료했다는 요청을 보내는데, 오리지널 레플리카에서 모든 세컨더리 레플리카들로부터 동기를 맞췄다는 요청을 받으면 syncStart의 값을 syncEnd 값으로 변경하고 'lock' 노드에 생성되었던 자식 노드를 없앰으로써 분산 락을 해제한다. Fig. 1(d)는 한 번의 메시지 동기화 과정이 끝날 때의 모습으로, 메시지 동기화가 끝난 후에는 모든 레플리카들은 syncStart 값인 오프셋 103을 가진 메시지까지 갖게 된다.

만약 메시지 동기화 중에 다른 세컨더리 레플리카가 메시지 동기화를 원한다면 lock 노드에 자식 노드를 생성한다. 그러나 이미 lock 노드에 자식 노드가 생성되어 있었다면 오리지널 레플리카는 lock의 자식 노드 생성으로 인해 발생한 알림을 통해 해당 세컨더리 레플리카가 하트비트를 했음을 확인하며, 해당 세컨더리 레플리카의 동기화는 현재 진행되고 있는 동기화 과정이 대체한다.

3.3 사용자 메시지 요청의 처리

본 절에서는 오리지널 레플리카가 사용자의 요청을 수용하고, 요청이 다른 세컨더리 레플리카들에 분산되어 처리되는 과정에 관해서 설명한다. 오리지널 레플리카를 통해 라운드 로빈하게 분산되는 요청은 오리지널 레플리카 또는 세컨더리 레플리카가 처리하게 된다. 아래의 Fig. 5-7에서 두 줄로 된 실선으로 표현된 브로커는 오리지널 레플리카이며, 한 줄로 된 실선으로 표현된 브로커는 세컨더리 레플리카이다.

1) 메시지 생산 요청의 처리

사용자의 메시지 생성 요청은 다음과 같은 순서로 처리된다. 먼저, (Fig. 2의 1번) 사용자가 처음 브로커와 메시지를 주고받을 때는 생산한 메시지를 보내기 위해 임의의 브로커 노드에 어떤 레플리카에 요청을 보내야 하는지에 대해 메타데이터 요청을 보낸다. (Fig. 2의 2, 3번) 메타데이터 요청을 받은 브로커 노드는 코디네이터에게 오리지널 레플리카의 정보를 받는다. (Fig. 2의 4번) 코디네이터로부터 받은 메타데이터는 사용자에게 전달된다. (Fig. 2의 5번) 메시지 생산 요청은 4번에서 받은 메타데이터를 통해 오리지널 레플리카에 보내진다. 메타데이터 요청을 받은 이후에 메시지 생산자는 오리지널 레플리카가 바뀌기 전까지 추가적인 메타데이터 요청 없이 계속해서 오리지널 레플리카에게 생산 요청을 보낸다. (Fig. 2의 6번) 생산 요청을 받은 오리지널 레플리카는 레플리카의 브로커 아이디 값이 작은 순서대로 라운드 로빈하게 요청을 분산시킨다. Fig. 2에서는 Broker1-Broker2-Broker3의 순서대로 요청이 분산된다. (Fig. 2의 7번) 요청 분산 후에 요청을 처리할 레플리카는 요청에 있는 메시지를 자신의 디스크에 저장한 후 메시지 생산자에게 ack를 돌려준다.

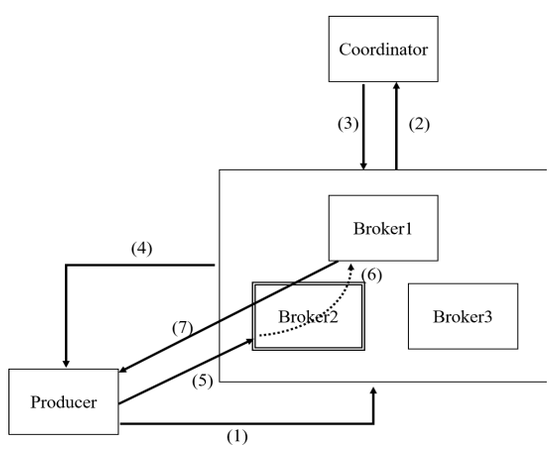

Fig. 2. Message Production Request Processing

2) 소비 요청 처리 과정

사용자의 메시지 생성 요청은 다음과 같은 순서로 처리된다. 먼저, (Fig. 3의 1번) 사용자가 처음 브로커와 메시지를 주고받을 때는 소비를 원하는 메시지를 받기 위해 임의의 브로커 노드에 어떤 레플리카에 요청을 보내야 하는지에 대해 메타데이터 요청을 보낸다. (Fig. 3의 2, 3번) 메타데이터 요청을 받은 브로커 노드는 코디네이터에게 오리지널 레플리카

의 정보를 받는다. (Fig. 3의 4번) 코디네이터로부터 받은 메타데이터는 사용자에게 전달된다. (Fig. 3의 5번) 메시지 소비 요청은 4번에서 받은 메타데이터를 통해 오리지널 레플리카에 보내진다. 메타데이터 요청을 받은 이후에 메시지 소비자는 오리지널 레플리카가 바뀌기 전까지 추가적인 메타데이터 요청 없이 계속해서 오리지널 레플리카에 소비 요청을 보낸다. (Fig. 3의 6번) 소비자에게 보낼 메시지가 가장 최근의 메시지 동기화 과정 이후에 처리된 메시지라면, 오리지널 레플리카는 다른 모든 세컨더리 레플리카들에 소비자의 요청을 보낸다. (Fig. 3의 7번) 오리지널 레플리카는 가장 최근에 동기화된 메시지까지를 보내고, 소비자의 요청을 전달받은 세컨더리 레플리카들은 자신이 가진 syncEnd 이후의 메시지들 중에서 소비자가 원하는 메시지를 보낸다.

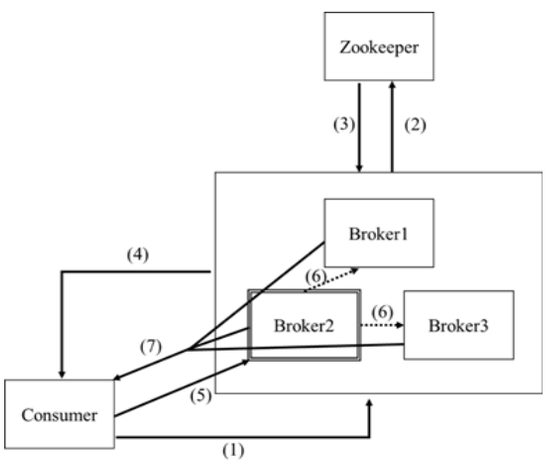

Fig. 3. Message Consumption Request Processing

4. 성능 평가

본 장에서는 액티브-액티브 구조를 효과적으로 적용하여 장애 극복 능력을 유지하면서도 높은 가용성을 얻을 수 있는 제안 기법의 효과를 평가하기 위해 액티브-스탠바이 구조를 활용하고 있는 대표적인 분산 메시지 브로커인 아파치 카프카에 대한 프로토타입 시스템과 제안 기법의 핵심 기능을 갖는 시스템을 구현하였으며, 이를 통해 분산 브로커에 사용자의 요청 부하가 발생했을 때 제안 기법의 효과를 알아보고자 하였다. 성능 평가에서는 메시지 처리 성능, 즉 각 사용자의 요청이 브로커에게 전달되어 처리된 후 응답을 받기까지의 처리 시간을 측정하여 비교하였으며, 처리 시간은 각각 메시지의 생산, 메시지의 소비 및 메시지의 생산과 소비의 세 실험 결과에 대해 각각 비교하였다. 또한 분산 락에 의한 오버헤드의 경우는 처리시간을 모델링하여 수식으로 제시하였다.

4.1 성능 평가 환경

본 실험을 위해 Amazon EC2 t2.medium 노드 4개를 사용하였으며, Table 1은 이의 구체적인 사양을 설명하고 있다. 메시지 브로커는 2개, 메시지 생산자와 메시지 소비자는 각 1개의 노드를 사용하였으며, 2개의 브로커 노드는 각각 액티브-액티브 레플리카와 액티브-스탠바이 레플리카를 구현하기 위해 사용되었다.

본 실험에서는 레플리카의 동기화의 시기를 일치시켜서 모든 레플리카들이 같은 주기에서 동기화를 수행하도록 하였다. 이 동기화 주기 자유도 여부는 3장에서 설명한 바와 같이 성능에는 큰 차이가 없지만, 액티브-스탠바이와 액티브-액티브를 동일한 환경에서 비교 평가하는 실험을 진행하기 위해서이다.

Table 1. Experimental Environment

	Amazon EC2 t2.medium
vCPU	2
RAM	4GB
OS	Ubuntu 16.04

Table 2는 실험에서 사용하는 데이터 구성, 즉 사용자가 생산하고 소비하는 메시지에 대한 상세 정보이다. 생산자가 만드는 메시지는 Project Gutenberg [14]에서 제공하는 문서 데이터의 일부를 사용하였으며, 소비자는 요청한 메시지를 받아서 해당 메시지에 관한 글자 수 카운트 예제를 진행하였다(Table 3 참조).

Table 2. Experimental Data Set

	Experimental data set
Dataset	'Alice's Adventures in Wonderland by Lewis Carroll' from Project Gutenberg [14]
File size	152,331 Byte

Table 3. Message Setup

Number of Topic	1
Partition number per Topic	1
Message size	100 Byte

4.2 성능 평가 결과 및 분석

1) 메시지 동기화 오버헤드

본 절에서는 같은 양의 메시지의 동기를 맞출 때 액티브-스탠바이 구조의 레플리카에서의 동기화 과정과 제안 기법에서 사용하는 액티브-액티브 구조의 레플리카에서의 동기화 과정에서 걸리는 오버헤드를 수식을 통해 비교하였다.

Table 4는 식에서 사용된 매개변수들에 대한 정보이다. n은 분산 브로커를 구성하는 레플리카의 수, m은 전체 레플리카에서 동기화가 필요한 메시지의 수, r_t는 네트워크를 통

Table 4. Description of Configure Parameter

Parameter	Description
n	Total number of replicas that make up the distributed broker
m	Total number of message that need to be synched
r_t	The time when one request is delivered (the time the request was received - the time the request was sent)
s_t	The time it takes to save one message(the time immediately after saveing a message to disk - the time just before saving a message to disk)
a	Parameter to consider the synchronization time of all standby replicas in the active-standby configuration $(1 \le a \le n-1)$
b	Parameter added for situations in which all replicas send and receive messages in parallel in the active-active architecture $(1 \le b \le 2)$

해 하나의 요청이 전달되는 시간으로 '(한 요청에 대한 응답을 받은 시간-한 요청을 보낸 시간)/2'를 사용하였고, s_t는 메시지 하나를 디스크에 저장하는 데 걸리는 시간으로 '디스크에 한 메시지를 저장한 후 시간-디스크에 한 메시지를 저장하기 전 시간'을 의미한다. r_t는 네트워크를 통해 요청을 보내는 것이므로 디스크에 접근하는 시간을 포함하는 s_t에 비해 항상 작은 값을 갖는다.

a) 액티브-스탠바이 구조에서의 동기화 과정 오버헤드

액티브-스탠바이 구조의 레플리카에서 동기화를 위해 걸리는 시간($time_{AS}$)은 Equation (1)과 같다. a는 모든 스탠바이 레플리카들의 동기화 시간을 고려하기 위한 매개변수로 한순간에 메시지 동기화를 진행하는 스탠바이 레플리카의 수를 의미한다. 같은 순간에 모든 스탠바이 레플리카의 동기를 맞춘다면 1을, 레플리카의 동기화 과정이 각각 따로 일어날 경우 $n-1$ 값을 가지며 레플리카들의 주기에 따라 1에서 $n-1$사이의 값을 가진다.

$$time_{AS} = am^{*}(2^{*}r_t + s_t) \quad (1)$$

b) 액티브-액티브 구조에서의 동기화 과정 오버헤드

액티브-액티브 구조에서 메시지는 전체 레플리카들에 $\lceil \frac{m}{n} \rceil$ 개 (메시지 수, 정수)씩 나뉘어 있으며 모든 레플리카가 메시지 동기화를 진행한다. 액티브-액티브 구조에서 동기화를 위해 걸리는 시간($time_{AA}$)은 Equation (2)와 같은데, b는 메시지를 주는 과정과 받는 과정이 병렬적으로 일어나는지에 관해 추가된 매개변수로, 메시지를 주고받는 과정이 동시에 일어날 경우 1을, 메시지를 주는 과정과 메시지를 받는 과정이 따로 일어나게 되면 2를 가지며 네트워크 상황에 따라 1에서 2 사이의 값을 갖게 된다.

$$time_{AA} = \left\{(n+1) + \left\lceil \frac{m(n-1)}{n} \right\rceil b\right\}^{*} r_t + \left\lceil \frac{m(n-1)}{n} \right\rceil {}^{*} s_t \quad (2)$$

c) 두 구조의 오버헤드 비교

$time_{AA}$와 $time_{AS}$값이 각각 1일 때는 $time_{AA}$가 $time_{AS}$보다 $(\frac{n-1}{n}+n)r_t \approx 2n \times r_t$만큼은 느리지만 $\left\lceil \frac{m}{n} \right\rceil s_t$만큼 빠른데, r_t가 항상 s_t보다 훨씬 작으며 레플리카 수 n은 메시지 수 m보다 작기 때문에 레플리카의 수가 월등히 많으면서 동기화되지 않은 메시지의 수가 적지 않다면 $time_{AA}$가 더 빠르다. 또한, $time_{AA}$와 $time_{AS}$값이 각각 $n-1$과 2로 최대일 때는 $time_{AA}$가 $(n-\frac{2}{n}-3)r_t + \frac{n-1}{n} s_t \approx n \times r_t + s_t$ 만큼 더 빠르다.

레플리카를 활용했을 때 발생하는 메시지 동기화 오버헤드는 위와 같이 수식으로 표현할 수 있으며, 이 수식에 나타나는 상수들의 값들은 조건에 따라 달라질 수 있다.

2) 메시지 처리 시간

본 절에서는 제안 기법을 적용한 분산 브로커의 가용성 향상 효과를 알아보기 위해 제안 기법을 적용한 경우와 그렇지 않은 경우의 분산 브로커의 메시지 처리 시간을 측정하여 비교하였다. 여기서 메시지 처리 시간은 사용자가 요청하는 메시지가 처리되기까지 걸리는 시간으로, 1) 생산자가 보낸 메시지가 브로커에 저장된 후 브로커가 생산자에게 응답하기까지 걸린 시간, 2) 소비자의 요청을 보내면 브로커가 요청에서 가리키는 메시지를 디스크에서 읽은 후 응답으로 메시지를 돌려주기까지 걸리는 시간, 3) 생산자가 만든 메시지가 브로커를 거쳐 소비자에게 전달되기까지 걸리는 시간으로 나누어 볼 수 있으며, 세 가지 경우에 대해 각각 실험을 진행하였다.

a) 생산자-브로커 메시지 처리 시간

Fig. 4는 생산자가 보낸 메시지가 브로커에 저장된 후 생산자가 응답을 받는 데에 걸리는 시간을 측정한 결과로, 실험 결과 제안 기법이 기존 기법에 비해 최소 약 1.34배에서 최대 약 1.9배의 처리 시간을 단축했음을 보인다. 메시지 생산자의 수

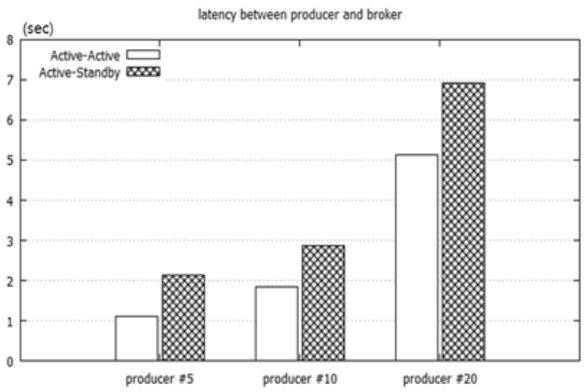

Fig. 4. Message Processing Time Between Producer and Broker

가 증가함에 따라 시간 단축의 폭이 감소한 원인으로는, 생산자가 많아지면서 작성되는 메시지의 양 역시 증가하였지만 브로커를 구성하는 레플리카의 수가 작아 요청이 충분히 분산되지 못했기 때문에 액티브-스탠바이 구조보다 제안 기법에서 동기화로 인한 오버헤드의 영향이 더 컸던 것으로 보인다.

b) 브로커-소비자 메시지 처리 시간

Fig. 5는 소비자의 요청을 통해 메시지가 브로커에서 소비자에게 전달된 후 메시지를 처리하기까지의 시간을 측정한 결과이다. 실험 결과 제안 기법이 모든 경우에서 기존의 액티브-스탠바이 구조 대비 약 1.7배의 처리 시간을 단축했다. 메시지 소비 작업만 진행될 때에는 생산 요청으로 인한 추가되는 메시지의 동기화 과정이 없어 두 구조 모두 동기화 오버헤드의 증감 요소가 없었으며, 따라서 제안 기법을 사용한 경우에 더 빠른 메시지 동기화가 이루어져 처리 시간이 향상된 것으로 보인다.

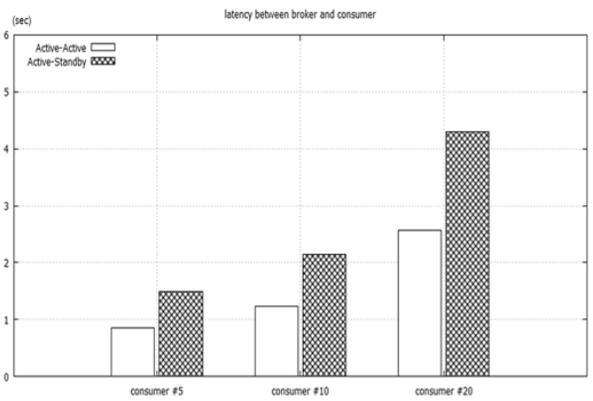

Fig. 5. Message Processing Time Between Consumer and Broker

c) 생산자-소비자 메시지 처리 시간

Table 5와 Fig. 6은 생산자가 보낸 메시지가 브로커에 저장된 후 소비자에게 전달되어 처리되기까지의 전체 시간을 측정한 결과이다. 실험 결과 제안 기법이 최소 약 1.11배에서 최대 약 1.33배의 처리 시간을 단축하였다. 앞 두 실험결과와 다르게 소비자의 처리 시간이 급격히 증가한 것은 메시지를 소비하는 과정이 메시지가 생산되고 다른 레플리카들에게 동기화되는 과정에 종속적이기 때문이다. 또한 브로커가 생산 요청과 패치 요청을 모두 처리하기 때문에 한 번의 동기화 과정에서 많은 수의 메시지를 동기화하지 못하여 메시지 동기화로 인한 오버헤드가 충분히 줄여지지 않은 것도 원인이다.

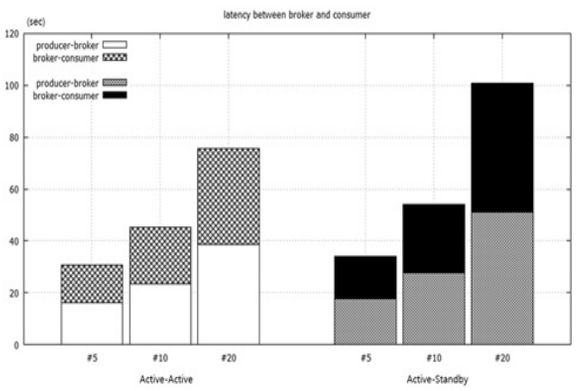

Fig. 6. Message Processing Time Between Producer and Consumer

5. 결론 및 향후 연구

본 연구에서는 장애 상황의 극복이 가능하면서도 분산 메시지 브로커의 가용성을 향상시키기 위해 메시지 레플리카를 액티브-액티브 구조로 구성하여 분산 브로커의 요청 부하를 분산시키는 기법을 제안하였다. 스탠바이 레플리카들이 액티

Table 5. Processing Time of Producer-Consumer Message

The number of producers and consumers	Active-Active		Active-Standby	
	Process time in producer (sec)	Process time in consumer (sec)	Process time in producer (sec)	Process time in consumer (sec)
5	15.9828	14.5984	17.5564	16.5332
10	23.2903	21.8962	27.7402	26.4027
20	38.68105	37.0657	51.0738	49.8743

브 레플리카로부터 요청을 전달받아 나누어 처리함으로써 브로커를 구성하는 노드 수의 증가 없이 요청 부하를 분산시킬 수 있었다. 이때 메시지 동기화 과정은 분산 코디네이터를 이용, 분산 락을 구현함으로써 모든 액티브 레플리카들이 한 때에 동기화를 진행하도록 하였고 각 액티브 레플리가 동기화를 할 때보다 추가적인 오버헤드를 적게 하였다.

실험에 앞서 레플리카들에서 메시지 동기화를 할 때 생기는 오버헤드를 수식으로 표현하여 액티브-액티브 구조의 레플리카를 활용하는 경우, 메시지의 수가 많아지더라도 액티브-스탠바이 구조의 레플리카보다 적은 오버헤드로 메시지의 동기를 맞출 수 있다는 것을 보였다. 실제 데이터와 구현한 프로토타입들을 이용해 실시한 실험의 결과를 통해 메시지의 생산/소비 요청과 메시지 동기화가 일어나는 상황에서 각 요청을 높은 성능으로 처리함을 보였다.

본 제안 기법에서는 동기화 체이닝 문제, 즉 동기화가 마무리되지 않은 레플리카를 동기화하기 위해 기다리는 다른 레플리카는 이전 레플리카의 동기화를 기다려야만 하는 문제는 여전히 해결하지 못하였다. 다른 분산 브로커 시스템에서도 아직 해결하지 못하고 있는 이 문제 역시 향후 연구를 통해 해결하도록 한다.

References

[1] P. T. Eugster, P. A. Felber, and R. Guerraoui, “The many faces of publish/subscribe,” *ACM Computing Surveys*, Vol.35, No.2, pp.114-131, 2003.

[2] E. Curry, “Message-oriented middleware,” in Middleware for communications, John Wiley & Sons, pp.1-28, 2004.

[3] G. Banavar, T. Chandra, R. Strom, and D. Sturman, “A case for message oriented middleware,” In *International Symposium on Distributed Computing*, pp.1-17, 1999.

[4] G. Pardo-Castellote, “OMG data-distribution service: Architectural overview,” In *Proc. IEEE International Conference on Distributed Computing Systems Workshops'03*, 2003, pp. 200-206.

[5] A. Fedoruk and R. Deters, “Improving fault-tolerance by replicating agents,” In *Proc. 1st International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2*, ACM, pp.737-744, 2002.

[6] The Apache Software Foundation, “Welcome to ApacheTM Hadoop!,” The Apache Software Foundation, 2014. [Online]. Available: http://hadoop.apache.org. [Accessed Nov. 30, 2017].

[7] The Apache Software Foundation, “Apache Hadoop 2.9.0 - HDFS High Availability,” The Apache Software Foundation, 2017. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html. [Accessed Nov. 30, 2017].

[8] J-H. Hwang, M. Balazinska, and A. Rasin, “High-availability algorithms for distributed stream processing,” In *Proc. 21st International Conference on Data Engineering'05*, pp. 779-790, 2005.

[9] NGINX Inc., “Load balancing with NGINX Plus,” NGINX Inc., 2017. [Online]. Available: https://www.nginx.com/products/nginx/load-balancing/#load-balancing-methods. [Accessed Nov. 30, 2017].

[10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “ZooKeeper: Wait-free Coordination for Internet-scale Systems,” In *Proc. USENIX Annual Technical Conference*, Vol. 8, pp.145-158, 2010.

[11] The Apache Software Foundation, “Samza,” The Apache Software Foundation. [Online]. Available: http://samza.apache.org. [Accessed Nov. 30, 2017].

[12] The Apache Software Foundation, “Apache Storm,” The Apache Software Foundation, 2015. [Online]. Available: http://storm.apache.org. [Accessed Nov. 30, 2017].

[13] The Apache Software Foundation, “Spark Streaming | Apache Spark,” The Apache Software Foundation. [Online]. Available: http://spark.apache.org/streaming. [Accessed Nov. 30, 2017].

[14] Project Gutenberg Literary Archive Foundation Inc., “Gutenberg,” Project Gutenberg Literary Archive Foundation Inc., 2017. [Online]. Available: https://www.gutenberg.org. [Accessed Nov. 30, 2017].

서 경 희

https://orcid.org/0000-0002-9570-4064

e-mail : algedian@ajou.ac.kr

2016년 아주대학교 정보컴퓨터공학과 (학사)

2018년 아주대학교 컴퓨터공학과(석사)

2018년~현 재 (주) NHN 사원

관심분야 : 병렬/분산 처리, 메시징 시스템

여 상 호

https://orcid.org/0000-0002-9194-7552

e-mail : soboru963@ajou.ac.kr

2017년 아주대학교 소프트웨어공학과 (학사)

2017년~현 재 아주대학교 컴퓨터공학과 석사과정

관심분야 : 심층강화학습, 분산시스템

오 상 윤

https://orcid.org/0000-0001-5854-149X

e-mail : syoh@ajou.ac.kr

2006년 미국 인디애나대학교 컴퓨터공학과 (박사)

2006년~2007년 SK텔레콤 전략기술부문

2007년~현 재 아주대학교 소프트웨어학과 교수

관심분야 : 분산/병렬 시스템, 고성능컴퓨팅, 클라우드컴퓨팅, Semantic Web