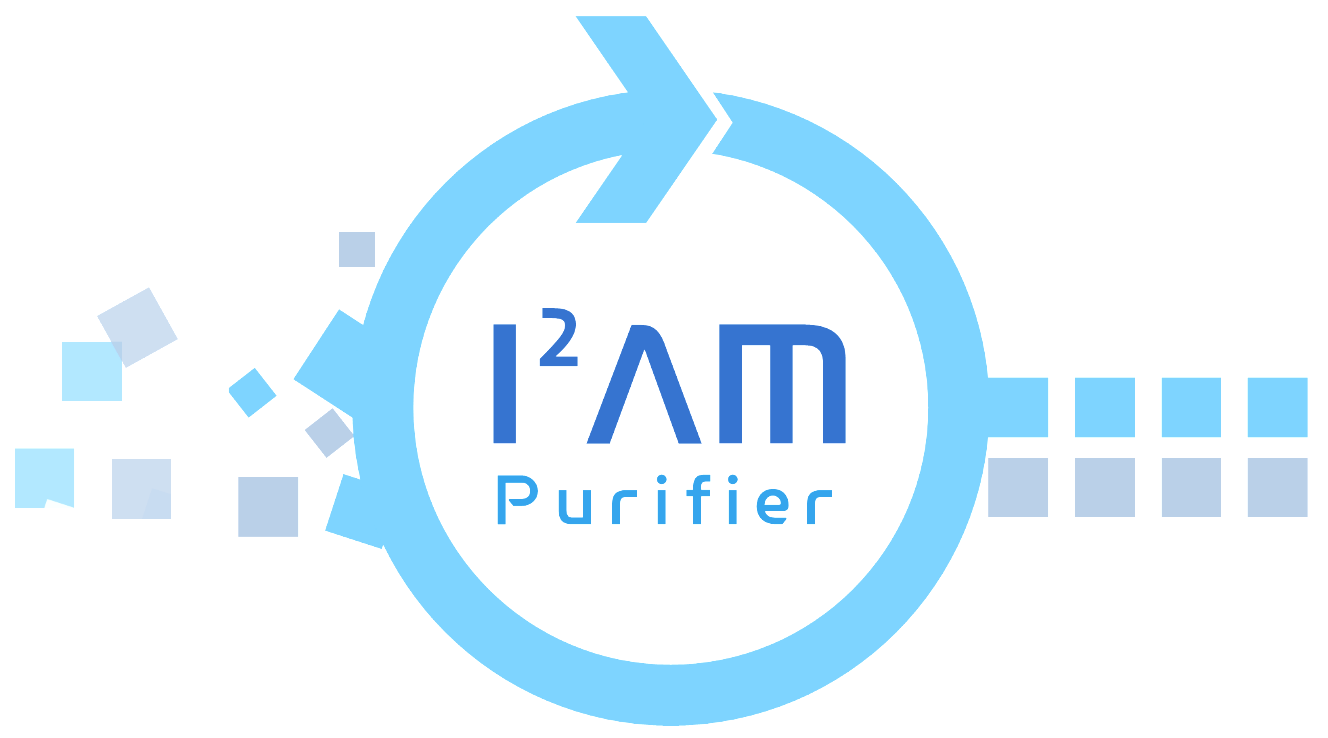
****

2016. 10.

강원대학교, 연세대학교

시스템 분석서

문서 정보

|  |  |
| --- | --- |
| **버 전** | 0.1 |
| **작성일** | 2016-11-01 |
| **상 태** | 🞎 완료 🞎 진행 중 🗹 초안 |
| **작성자** | 이름(소속) |
| **검토자** | 이름(소속) |
| **승인자** | 이름(소속) |

**목 차**

[시스템 분석서 1](file:///D:\DKE\1_I2AM\보고서_양식.docx#_Toc466993626)

[문서 정보 2](#_Toc466993627)

[1. 아파치 스톰 1](#_Toc466993628)

[1.1. 개요 1](#_Toc466993629)

[1.2. 구성요소 1](#_Toc466993630)

[1.3. 병렬성 1](#_Toc466993631)

[1.4. 그룹핑 1](#_Toc466993632)

[1.5. 관리 명령어 1](#_Toc466993633)

# 아파치 스톰

## 개요

스톰은 자바(java)와 클로저(clojure)로 구현된 실시간 분산처리 시스템이다. 스톰은 크게 님버스(Nimbus), 슈퍼바이저(Supervisor) 데몬 프로세스로 구성되어있으며, 이 두 데몬의 상태 정보를 공유하기 위해 주키퍼(Zookeeper) 데몬 프로세스를 추가적으로 사용한다. 또한 실질적인 처리를 위한 구성 요소로 워커(Worker), 익스큐터(Excutor), 토폴로지(Topology), 스트림(Stream), 스파웃(Spout)과 볼트(Bolt)라는 작업단위가 있다. 주요 인터페이스는 자바로 정의되어 있고 핵심 로직은 거의 클로저로 구현되었다.

스톰은 확장성, 장애 대응성, 메시지 전달 보장, 여러 프로그래밍 언어 지원, 다양한 시스템 연계, 오픈 소스라는 특징이 있으며, 다음은 이러한 특징에 대한 자세한 설명이다.

* + 확장성  
    스톰은 수평적으로 노드의 추가 또는 제거가 가능하다.
  + 장애 대응성  
    주키퍼를 통해서 전체 클러스터의 운영 상태를 모니터링한다. 이때, 특정 노드에서 장애가 발생하더라도 시스템이 전혀 문제 없이 작업을 진행할 수 있도록 장애가 발생한 노드에 할당된 작업은 다른 노드에 할당해서 처리하고, 장애 노드에 대해서는 복구처리를 자동으로 수행해준다.
  + 메시지 전달 보장  
    장애가 발생하던 간에 메시지의 유실 없이 최소한 한 번 메시지가 처리될 수 있게 지원한다.
  + 여러 프로그래밍 언어 지원  
    스톰은 기본적으로 JVM(Java virtual machine) 위에서 동작한다. 하지만 Twitter의 Thrift 프로토콜을 기반으로 동작 하기 때문에, Java 뿐만 아니라 JVM을 사용하지 않는 경우에 대해서는 stdin/stdout을 통해서 데이터를 주고 받음으로써 Ruby, Python, Javascript, Perl 등 다양한 언어를 사용할 수 있다.
  + 다양한 시스템 연계  
    스톰은 다른 다양한 솔루션과 통합이 가능하다. 데이터를 수집하는 부분에서는 Kestrel, RabbitMQ, Kafka, KMS 등이 연동 가능하고 RDBMS, Cassandra, MongoDB 등 다양한 데이터 베이스에도 쉽게 연동이 가능하다. 또한, CEP(complex event processing) 이벤트 처리 분야에서는 Drools, Esper 등과 연계가 가능하고, 그외에도 Elastic Search, node.js 등 다양한 솔루션과 연동을 통해서 시스템을 확장해 나갈 수 있다.
  + 오픈소스  
    스톰은 Apache License 2.0을 따르는 오픈소스 프로젝트이다. 따라서 개발자는 스톰을 쉽게 커스터마이징해서 사용할 수 있다.

## 구성요소

* + 님버스
    - 스톰 클러스터에 오직 하나만 존재하는 마스터 노드(master node)이다.
    - 클러스터에서 실행되는 토폴로지를 관리, 조절, 모니터링한다.
    - 사용자가 패키징된 토폴로지 JAR 파일과 해당 토폴로지 설정을 님버스 서버로 제출(submit)하면, 님버스가 필요한 수만큼의 슈퍼바이저에게 해당 JAR 파일을 전송한다.
    - 님버스는 슈퍼바이저 노드들에게 토폴로지 아카이브를 전송한 후, 각 슈퍼바이저에게 작업단위를 할당하고, 할당한 작업단위를 실행할 워커를 생성하라고 슈퍼바이저에게 요청한다.
    - 님버스는 하트비트(heartbeat)를 통해서 모든 슈퍼바이저의 상태를 관리하는데, 이를 받지 못하거나 해당 슈퍼바이저가 서비스 불능상태라는 것을 발견하면 그 슈퍼바이저가 하던 작업단위를 회수하여 클러스터 내의 다른 정상 동작 중인 슈퍼바이저에게 재할당한다.
  + 슈퍼바이저
    - 스톰 클러스터에서 머신 마다 하나로, 다수 존재하는 슬레이브 노드(slave node)이다.
    - 님버스로부터 작업단위를 할당 받고, 작업단위를 실행하는 워커를 생성 후 모니터링한다.
    - 슈퍼바이저가 생성한 워커에 에러가 발생해 예기치 않게 종료되었을 때 슈퍼바이저는 워커를 재생성한다.
  + 주키퍼
    - 주키퍼는 몇 가지 기본 기능과 그룹 기능을 제공하는 분산환경용 중앙 정보 저장소 서비스이다. 주키퍼는 간단하지만 강력한 분산 동기화 기능을 제공한다.
    - 스톰은 클러스터 내의 님버스와 슈퍼바이저 간에 작업단위 할당 정보, 워커 상태 정보, 토폴로지 메트릭과 같은 상태 정보를 공유하기 위해 주키퍼를 사용한다.
    - 님버스와 슈퍼바이저 간의 통신은 주키퍼의 상태 조작과 알림 구독을 통해 이루어 진다.

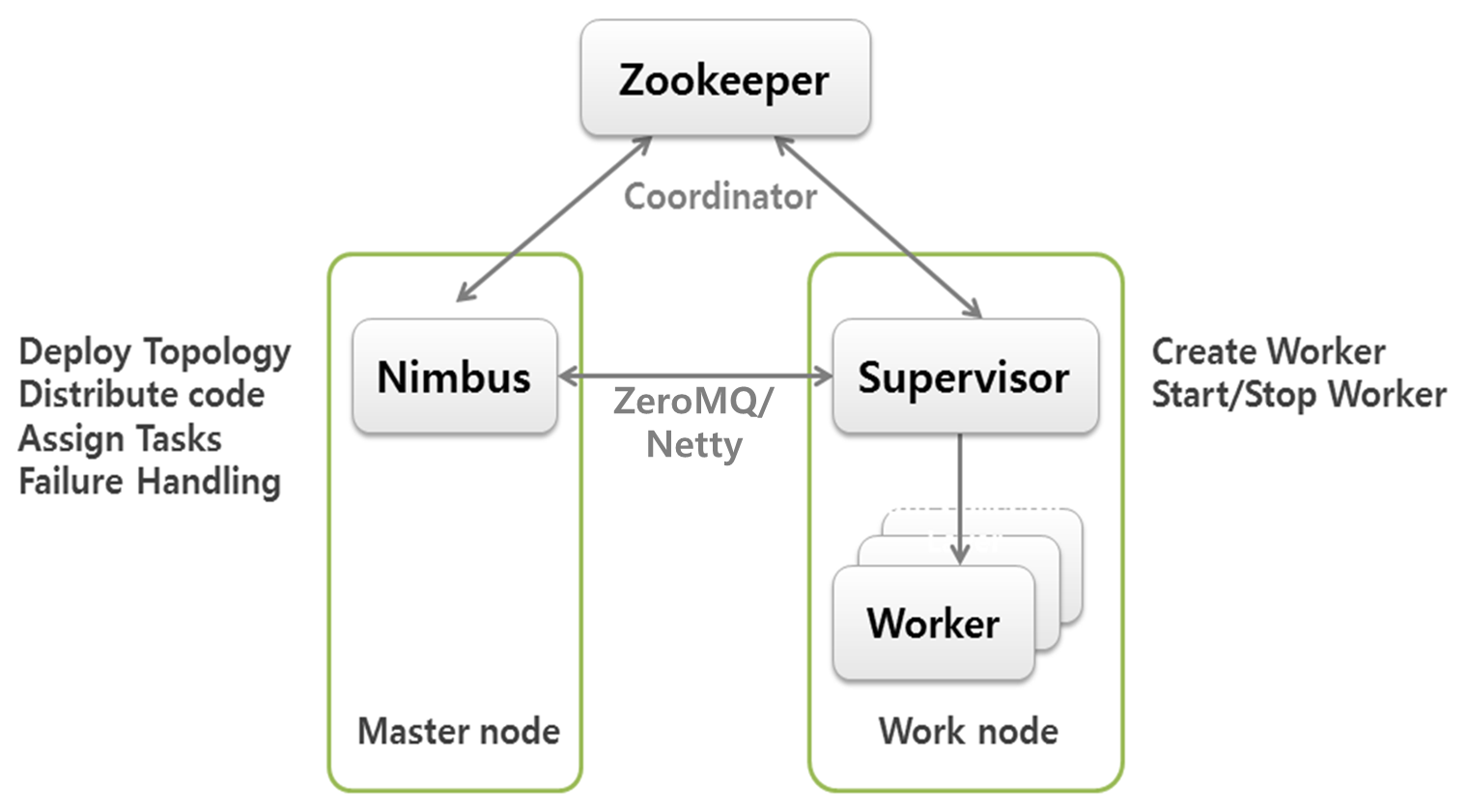


Figure 1 스톰의 구조.

## 병렬성

* + 워커
    - 슈퍼바이저가 있는 노드에서 동작하는 독립적인 JVM 프로세스이다.
    - 각 슈퍼바이저 노드는 하나 이상의 워커가 동작하도록 설정되어 있다.
    - 토폴로지는 하나 이상의 워커에서 동작하도록 배치된다.
  + 익스큐터
    - 워커 JVM 프로세스에서 동작하는 자바 스레드이다.
    - 단일 익스큐터에 다수의 작업단위를 배치할 수 있다.
    - 소스코드 상에서 재정의하여 명시하지 않으면 기본적으로 각 익스큐터는 한 개의 작업단위만 할당 받는다.

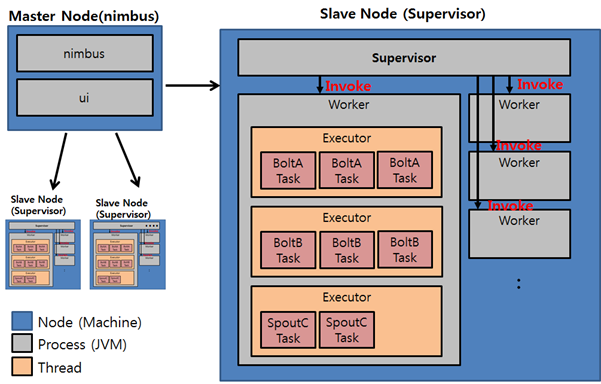
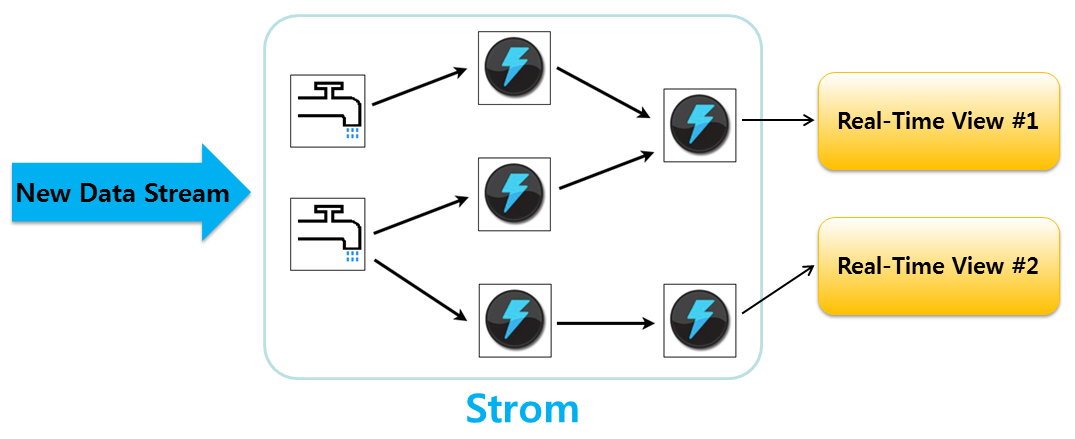


Figure 2 스톰 병렬구조.

* + 토폴로지
    - 스톰의 분산 연산 구조로 데이터 스트림, 스파웃, 볼트로 구성되어 있다.
    - 하둡과 같은 배치 처리 시스템의 잡과 거의 비슷하지만 배치 잡은 연산의 처음과 끝지점이 명확하게 정의되어 있는 반면 스톰 토폴로지는 정지 명령을 내릴 때까지 계속 동작한다.
  + 스트림
    - 스톰은 튜플이라는 key-value pair의 목록을 기본 데이터 구조로 갖고, 연속된 튜플들을 스트림이라 한다.
  + 스파웃
    - 데이터가 스톰 토폴로지로 들어가는 입구이다.
    - 데이터 소스와 연결을 맺고, 데이터를 튜플로 변환하여 스트림으로 튜플을 내보낸다.
    - 스톰은 스파웃을 구현할 때 필요한 간단한 API를 제공하고, 스파웃을 개발하는 것은 데이터를 가져오기 위해 데이터 소스의 API를 호출하는 것이 구현의 거의 대부분이다.
    - 데이터 소스로는 웹 기반 또는 모바일 애플리케이션의 클릭 스트림, 트위터나 소셜 네트워크 피드, 센서 출력, 애플리케이션 로그 이벤트 등이 있다.
  + 볼트
    - 스파웃 또는 다른 볼트로부터 데이터를 받아 실제 처리 로직을 구현할 수 있다.
    - 처리한 데이터를 선택적으로 하나 이상의 스트림으로 내보낼 수 있다.
    - 상상할 수 있는 모든 처리를 볼트로 구현할 수 있다. 일반적으로, 튜플 필터링, 조인과 합계 연산, 데이터베이스 읽기/쓰기가 있다.

 Figure 3 토폴로지 구성.

## 그룹핑

* + 스톰은 몇가지 스트림 그룹핑 기능을 제공함으로써 작업단위 사이의 데이터 흐름을 정의할 수 있다.
    - 셔플 그룹핑(Shuffle grouping): 타겟 볼트 작업단위들에게 무작위 순서로 튜플을 나누어 준다. 각각의 볼트가 받는 튜플의 수는 동일하다.
    - 필드 그룹핑(Field grouping): 그룹핑에서 정의한 필드의 값에 따라 튜플이 전달될 볼트가 정해진다. 예를 들어 스트림이 word 필드로 그룹핑되었다면 word 필드 값이 동일한 튜플들은 항상 동일한 볼트 작업단위로 전달된다.
    - 올 그룹핑(All grouping): 튜플 스트림의 복사본들이 모든 볼트 작업단위로 전달된다. 각각의 작업단위는 튜플의 복사본을 받는다.
    - 글로벌 그룹핑(Global grouping): 스트림의 모든 튜플을 단일 작업단위로 전달한다. 이 때 작업단위 아이디 값이 가장 작은 작업단위로 전달한다. 글로벌 그룹핑을 사용하는 볼트의 병렬화 지수를 늘리거나 작업단위의 수를 늘리는 것은 아무런 의미 없는 행동이다. 모든 튜플을 단일 JVM 인스턴스에게 전달하기 때문에 병목 현상이 발생할 수 있다.
    - 다이렉트 그룹핑(Direct grouping): 소스 스트림이 emitDirect() 메소드를 호출하여, 튜플을 내보내고 싶은 컴포넌트를 직접 지정할 수 있다. 다이렉트 스트림으로 선언된 스트림에서만 사용할 수 있다.
    - 로컬 또는 셔플 그룹핑(Local or shulffle grouping): 로컬 또는 셔플 그룹핑은 셔플 그룹핑과 비슷하지만, 같은 워커 프로세스에서 동작하는 볼트 작업단위 대상으로만 튜플을 셔플링한다. 그 외 기능은 셔플 그룹핑과 동일하다. 토폴로지의 성능을 네트워크 전송 한계까지 올릴 수 있다.

## 관리 명령어

* + 스톰의 관리 명령어를 이용해 클러스터에 토폴로지를 배포하거나 실행 중인 토폴로지를 관리할 수 있다. 관리 명령어는 보통 스톰 클러스터가 아닌 외부 워크스테이션에서 사용되고, 이 명령어는 님버스와 쓰리프트를 이용해 통신한다. 주요 명령어는 다음과 같다.
    - jar

jar 명령어는 클러스터에 토폴로지를 배포하기 위해 사용한다. topology\_class의 main() 메소드를 주어진 인자로 실행하고, topology\_jar\_path의 jar 파일을 클러스터 전체에 배포하기 위해 님버스로 jar 파일을 업로드한다. 배포가 끝나면 스톰은 토폴로지를 활성화하고, 토폴로지에 기재되어 있는 처리 로직을 시작한다.

사용법: storm jar topology\_jar\_path topology\_class [arg1 arg2 …]

* + - kill

kill 명령어는 배포된 토폴로지를 종료시킬 때 사용된다. kill 명령어는 인자로 받은 topology\_name에 해당하는 토폴로지를 종료시킨다. 스톰은 토폴로지의 스파웃이 데이터 소스로부터 이미 받은 튜플들이 안전하게 다 처리될 수 있도록 기다려야 하기때문에, 먼저 토폴로지의 스파웃을 topology.message.timeout.sec에 설정된 시간만큼 비활성화시킨다. 그런다음 워커들을 중단시키고 모든 저장된 상태 정보를 삭제한다. –w 인자를 사용해 topology.message.timeout.secs의 값을 재정의할 수 있다.

사용법: storm kill topology\_name [-w wait\_time]

* + - deactivate

deactivate 명령어는 토폴로지의 스파웃이 튜플 스트림을 생성하는 것을 멈추게 한다.

사용법: storm deactivate topology\_name

* + - activate

activate 명령어는 토폴로지의 스파웃이 튜플 스트림을 다시 생성하게 하는 명령어이다.

사용법: storm activate topology\_name

* + - rebalance

rebalance 명령어는 토폴로지를 종료하거나 재배포하지 않고 클러스터의 워커들에게 할당된 작업단위들을 재분배하는 명령어이다. 새로 추가된 노드는 새로운 토폴로지를 클러스터로 배포하지 않으면 아무런 작업단위도 할당받지 못하기 때문에, rebalance 명령어를 실행야한다. –n 인자를 이용해 토폴로지에 할당된 워커의 수를 변경하거나, -e 인자를 이용해 작업단위에 할당된 익스큐터의 수를 변경할 수 있다. rebalance 명령어를 실행하면 먼저 이미 토폴로지로 받은 튜플들이 모두 처리되기를 설정된 시간만큼 기다린 다음 슈퍼바이저 노드들에게 균등하게 워커를 재분배한다. 재분배가 끝나면 이전의 상태, activate 혹은 deactivate 상태로 돌려놓는다.

사용법: storm rebalance topology\_name [-w wait\_time] [-n worker\_count] [-e component\_name=executor\_count]…