SmartX mini labs 최종 과제 리포트

20185152 임민택

이번 실험에서는 kafka를 이용하여 zookeeper, broker 클러스터를 구축하고 producer와 consumer를 연결하여 데이터를 발생시키고 데이터베이스에 저장한 뒤 Grafana로 이를 시각화 해서 관찰하는 것을 목표로 한다. Kafka는 메시지 큐로 작동하는데 producer가 데이터를 보내면 이것을 큐에 저장한 뒤 consumer가 사용할 수 있도록 잠시 저장해두는 역할을 한다. InfluxBD는 consumer로 데이터를 받은 뒤 데이터베이스에 이를 저장하는 역할을 하고 당연히 kafka 서버와 연결 되어있어야 한다. 이 작업은 consumer.py에 관련된 코드들이 있었고 kafka 서버와의 연결과 메시지를 parsing하는 코드의 작성이 필요하였다. Producer들은 tcpdump라는 패킷 관찰 도구를 통해서 정보를 생성하고 전달하게 된다. 이것은 producer.py에 관련 코드들이 파이썬 코드로 작성되어있고 kafka 서버와의 연결 작업만 필요했다. Kafka 클러스터는 zookeeper와 broker로 구성되는데 이 둘은 localhost의 포트로 연결 되어있다. Zookeeper가 broker들을 관리하여 데이터를 분산해서 보내주는 역할을 주로 하고 broker가 실제적인 메시지 큐의 역할을 하게 된다. 이들을 실행하는 컨테이너들은 java 관련 패키지들이 설치 되어있어야 서버 구축이 가능하다. 마지막으로 InfluxBD에 쌓인 데이터들을 Grafana를 이용하여 시각화 하는 작업을 진행하면 최종적인 목표를 달성하게 된다.

전체적인 실험의 과정은 Lab3 Tower lab의 내용과 비슷한 면이 있었다. Kafka 서버를 구축하고 producer와 consumer를 연결하여 데이터를 발생시키고 저장한 것을 tower을 통해서 모니터링 하는 방식은 동일했다. 차이점은 Tower lab에서는 데이터의 생성을 flume 컨테이너로 진행하였고 이미 제공된 모듈을 활용하여 별도의 추가적인 연결 과정이나 python 코드 편집이 필요하지 않았다. Consumer의 연결 과정은 비슷했지만 tower 랩에서는 kafka 서버와의 연결 작업만 필요로 하였지만 이번 실습에서는 데이터를 정리하는 작업을 위한 코드를 작성하여야 했다. 실습에 필요한 모든 컨테이너들은 직접 작성한 Dockerfile로부터 생성한 이미지를 활용하여 했는데 이는 kafka의 버전이 달라서 발생하는 문제와 kafka 서버를 구축할 때 네트워크 관련된 제한이 있어서 이렇게 하게 되었다. 이 과정 중에 발생하는 문제들을 하나하나 원인을 찾고 해결하는 것에 상당히 많은 어려움이 있었고 Dockerfile 작성과 관련 패키지 설치하는 분야에서 많은 지식이 필요해 보였다.

실습을 진행하기 위해 가장 먼저 필요한 작업은 Dockerfile을 작성하는 것이다. Producer 컨테이너를 3개 구동시켜야 하는데 이들의 이미지를 생성해줘서 producer.py를 실행할 수 있게 만들어

주어야한다. 또한 서론에서 언급하였듯이 이후 진행 과정에서 kafka의 버전이 달라서 문제가 발생하기 때문에 kafka 서버를 구축하는 컨테이너 역시 직접 만든 이미지로 컨테이너를 띄워야 했기 때문에 Dockerfile에는 python 실행 관련 패키지와 java 관련 패키지, ping과 네트워크 툴, vim, git, tcpdump의 설치가 포함되어있다. 또 producer.py를 각 컨테이너 마다 작성하는데 시간이 많이 소요되므로 미리 작성하여서 이미지로 배포하였다. 각 컨테이너에서 이를 받아 ip만 변경해주면 될 것이다. Dockerfile의 내용은 아래와 같다.

```
FROM ubuntu:18.04
MAINTAINER lim, min teak

RUN apt-get update
RUN apt-get install -y software-properties-common
#RUN add-apt-repository -y ppa:fkrull/deadsnakes
#RUN apt-get update
RUN apt-get install -y --no-install-recommends python3.6 python3.6-dev python3-pip python3-setuptools python3-wheel gcc
#RUN apt-get install -y git

RUN python3.6 -m pip install pip --upgrade
RUN pip install kafka-python

RUN apt-get install -y iputils-ping
RUN apt-get install -y net-tools
RUN apt-get install -y tcpdump
RUN apt-get install -y vim
RUN apt-get install -y git


#Install Oracle JAVA
RUN  mkdir -p /opt
ADD jdk-8u172-linux-x64.tar.gz /opt
```

#Configurate environmental variables

ENV JAVA_HOME /opt/jdk

ENV PATH $PATH:/opt/jdk/bin

RUN ln −s ls /opt | grep "^jdk.*" /opt/jdk && ln −s /opt/jdk/bin/java /usr/local/bin/java

#this

#RUN mkdir −p /kafka

#WORKDIR /kafka

COPY kafka_2.12−2.4.0 /kafka

WORKDIR /kafka

COPY producer.py /kafka/producer.py

COPY consumer.py /kafka/consumer.py

WORKDIR /kafka

Dockerfile이 있는 디렉토리에는 다음과 같은 파일들이 들어있다.

```
victorlim@victorlimnuc:~/cstl_2020/kafka$ ls
consumer.py  jdk-8u172-linux-x64.tar.gz  kafka_2.12-2.4.0  zookeeper
Dockerfile   kafka                       producer.py
```

이 Dockerfile을 바탕으로 docker image를 생성하기 위해서는

sudo docker build --tag made .

명령어를 사용한다.

Producer의 역할을 할 컨테이너들을 A, B, C 3개를 작동시키는데

$sudo docker run -it --net=none --name con# made

명령어를 사용하였고

$sudo ovs-docker add-port br0 eno1 con# --ipaddress=192.168.100.#/24 --gateway=192.168.100.1

명령어를 이용하여 수동으로 ip를 설정해주고 네트워크를 연결해주었다. 이것은 이후에 ping을

보낼 ip를 명확하게 알고 모니터링 할 ip를 쉽게 지정하기 위함이다. 각 컨테이너와 nuc의 네트워크 설정은 ifconfig 명령어를 통해 볼 수 있고 이는 net-tools를 설치하면 실행이 가능하다. 할당된 ip는 아래와 같다.

```
root@af34c36b4a09:/kafka# ifconfig
eno1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.100.10  netmask 255.255.255.0  broadcast 0.0.0.0
        ether a2:e0:e4:c1:d4:85  txqueuelen 1000  (Ethernet)
        RX packets 56039  bytes 8011274 (8.0 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 56317  bytes 4213567 (4.2 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 4  bytes 200 (200.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4  bytes 200 (200.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@af34c36b4a09:/kafka#
```

conA의 ifconfig 결과

```
root@b92c590135b8:/kafka# ifconfig
eno1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.100.11  netmask 255.255.255.0  broadcast 0.0.0.0
        ether 7e:62:73:35:d9:62  txqueuelen 1000  (Ethernet)
        RX packets 1436  bytes 188329 (188.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1482  bytes 105977 (105.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

conB의 ifconfig 결과

```
root@f095ea4dfd4c:/kafka# ifconfig
eno1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.100.12  netmask 255.255.255.0  broadcast 0.0.0.0
        ether 26:70:16:e8:b8:78  txqueuelen 1000  (Ethernet)
        RX packets 28  bytes 3190 (3.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4  bytes 280 (280.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@f095ea4dfd4c:/kafka#
```

conC의 ifconfig 결과

```
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.100.1  netmask 255.255.255.0  broadcast 192.168.100.255
        inet6 fe80::d059:62ff:fe06:1a4a  prefixlen 64  scopeid 0x20<link>
        ether d2:59:62:06:1a:4a  txqueuelen 1000  (Ethernet)
        RX packets 257426  bytes 14956810 (14.9 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 416610  bytes 487096593 (487.0 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

br1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::38b1:31ff:fe33:444e  prefixlen 64  scopeid 0x20<link>
        ether 3a:b1:31:33:44:4e  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 24  overruns 0  frame 0
        TX packets 185  bytes 17214 (17.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

cali25d4f9721fc: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1440
        inet6 fe80::ecee:eeff:feee:eeee  prefixlen 64  scopeid 0x20<link>
        ether ee:ee:ee:ee:ee:ee  txqueuelen 0  (Ethernet)
        RX packets 581714  bytes 47845241 (47.8 MB)
```

Nuc의 ifconfig 결과(1)

```
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        inet6 fe80::42:6fff:fece:65ec  prefixlen 64  scopeid 0x20<link>
        ether 02:42:6f:ce:65:ec  txqueuelen 0  (Ethernet)
        RX packets 46581  bytes 2724010 (2.7 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 101871  bytes 154978074 (154.9 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eno1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.0.100  netmask 255.255.255.0  broadcast 192.168.0.255
        inet6 fe80::96c6:91ff:fea5:f09d  prefixlen 64  scopeid 0x20<link>
        ether 94:c6:91:a5:f0:9d  txqueuelen 1000  (Ethernet)
        RX packets 1892599  bytes 2459821600 (2.4 GB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1158734  bytes 104457855 (104.4 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 16  memory 0xc0b00000-c0b20000
```

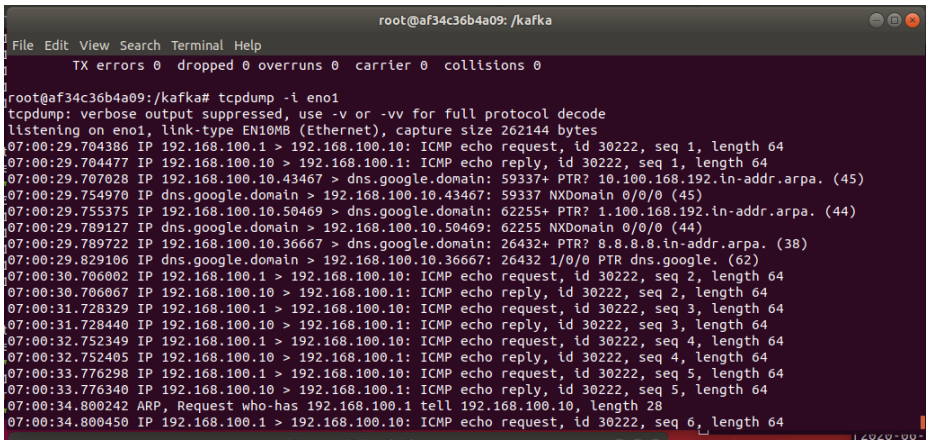
Nuc의 ifconfig 결과(2)

다음으로 consumer에 활용될 InfluxBD 관련 패키지 설치이다. 이후 실습에서 kafka 서버로부터 데이터를 가져와서 저장하는 역할을 수행하게 된다. 이 과정은 가이드라인에 모든 과정이 설명되어있어서 결과만 첨부한다.

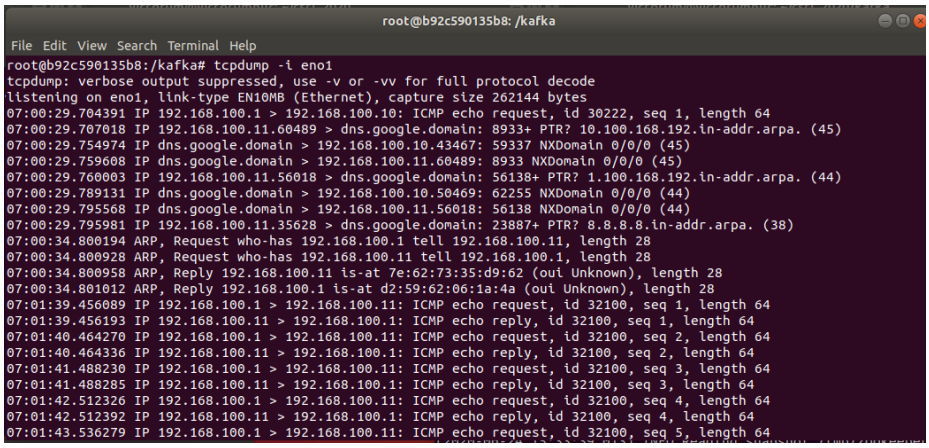
```
File  Edit  View  Search  Terminal  Help
6월 25 15:39:44 victorlimnuc influxd[1512]: [httpd] 127.0.0.1 - - [25/Jun/2020:
6월 25 15:47:13 victorlimnuc influxd[1512]: [retention] 2020/06/25 15:47:13 ret
..skipping...
● influxdb.service - InfluxDB is an open-source, distributed, time series databa
   Loaded: loaded (/lib/systemd/system/influxdb.service; enabled; vendor preset:
   Active: active (running) since Mon 2020-06-22 10:47:13 KST; 3 days ago
     Docs: man:influxd(1)
 Main PID: 1512 (influxd)
    Tasks: 11 (limit: 4915)
   CGroup: /system.slice/influxdb.service
           └─1512 /usr/bin/influxd -config /etc/influxdb/influxdb.conf

6월 25 15:37:44 victorlimnuc influxd[1512]: [httpd] 127.0.0.1 - - [25/Jun/2020:
6월 25 15:37:59 victorlimnuc influxd[1512]: [query] 2020/06/25 15:37:59 SHOW ME
6월 25 15:37:59 victorlimnuc influxd[1512]: [httpd] 127.0.0.1 - - [25/Jun/2020:
6월 25 15:38:16 victorlimnuc influxd[1512]: [query] 2020/06/25 15:38:16 SHOW DA
6월 25 15:38:16 victorlimnuc influxd[1512]: [httpd] 127.0.0.1 - - [25/Jun/2020:
6월 25 15:38:18 victorlimnuc influxd[1512]: [query] 2020/06/25 15:38:18 SHOW ME
6월 25 15:38:18 victorlimnuc influxd[1512]: [httpd] 127.0.0.1 - - [25/Jun/2020:
6월 25 15:39:44 victorlimnuc influxd[1512]: [query] 2020/06/25 15:39:44 SHOW DA
6월 25 15:39:44 victorlimnuc influxd[1512]: [httpd] 127.0.0.1 - - [25/Jun/2020:
6월 25 15:47:13 victorlimnuc influxd[1512]: [retention] 2020/06/25 15:47:13 ret
```

systemctl status influxdb 결과

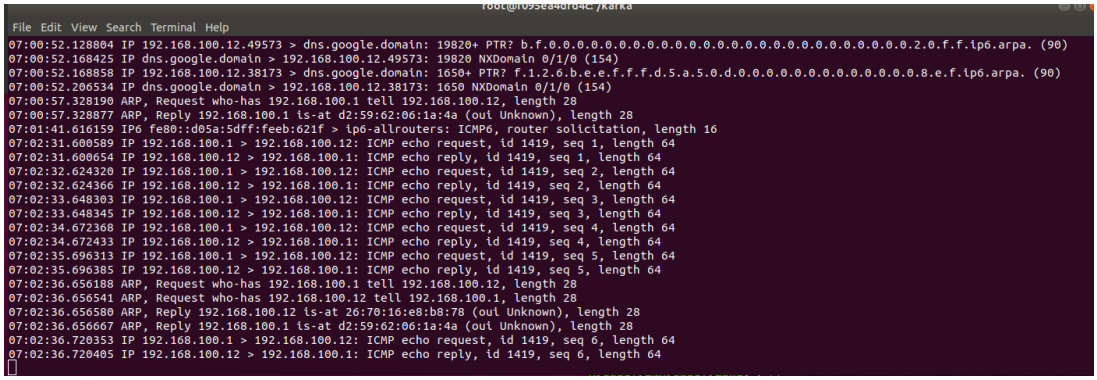
이제 각 producer 컨테이너들에서 tcpdump를 실행하고(이미지에 포함되어서 설치되어있다) nuc에서 ping을 보내서 이를 확인해본다. tcpdump –i eno1 명령어를 사용하였고 eno1을 통해 이동하는 모든 패킷을 관찰할 수 있었다. tcpdump가 그 인터페이스를 모니터링 하는 도구이기 때문에 실행 후 ping을 보내서 패킷을 주고 받아야 관련 데이터가 뜨게 된다.



conA에서 tcpdump 결과



conB에서 tcpdump 결과



conC에서 tcpdump 결과

다음은 kafka 클러스터 서버의 구축이다. https://github.com/gist-banana/cstl_2020에 있는 kafka 디렉토리 안의 kafka 소스를 활용하여 컨테이너를 작동시켰는데(작성한 Dockerfile로 생성된 이미지로) 네트워크를 none으로 설정하고 수동으로 ip를 할당해주면 zookeeper와 broker의 연결에 어려움이 있었다. 이는 zava 코드에서 localhost를 계속해서 찾는 이유에서 발생 한 오류로 보이는데 이를 해결하는 것보다 컨테이너의 네트워크를 host로 설정하고 –p 옵션을 활용하여 외부에서 포트로 접근할 수 있도록 하는 방법을 선택하였다.

$sudo docker run -it --net=host –p 9092:9092 --name brokernew made

$sudo docker run -it --net=host –p 2181:2181 --name zookeepernew made

명령어를 각각 사용하였다. 이제 클러스터링을 위해서 각 컨테이너에서 zookeeper.properties와 server.properties를 확인하고 연결 가능하도록 설정해준다. 디폴트 값을 사용하였기 때문에 별도로 변경한 값은 없었다. 이후

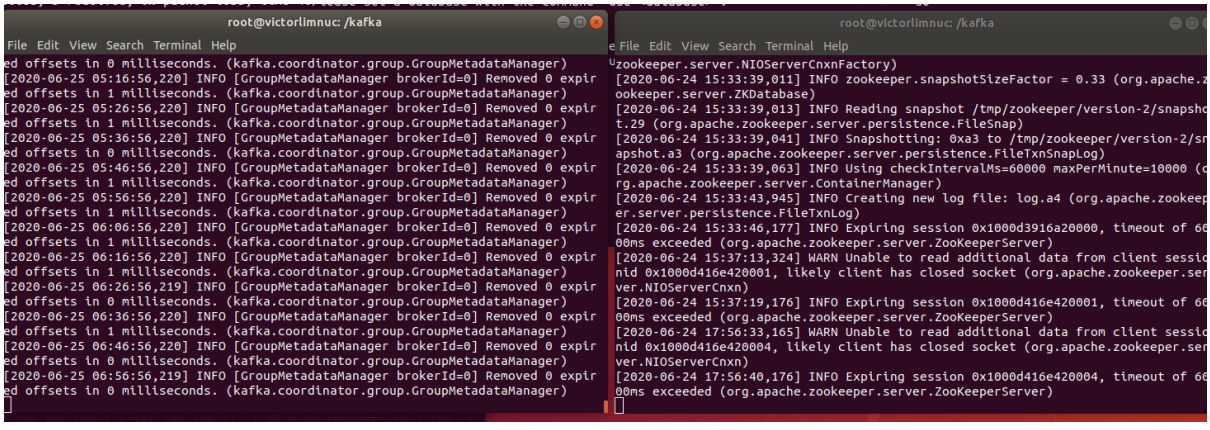
bin/zookeeper-server-start.sh config/zookeeper.properties

bin/kafka-server-start.sh config/server.properties &

를 각각 실행시켜 둘을 연결시키고 클러스터 서버를 구축하였다. 또한 이 서버에 topic을 만들어주기 위해서 broker를 잠시 백그라운드로 실행시키고

bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 3 --topic cstl

명령어를 실행하여 cstl이라는 topic을 만들어준다. 이 topic의 이름은 producer.py에서 참고하였다.

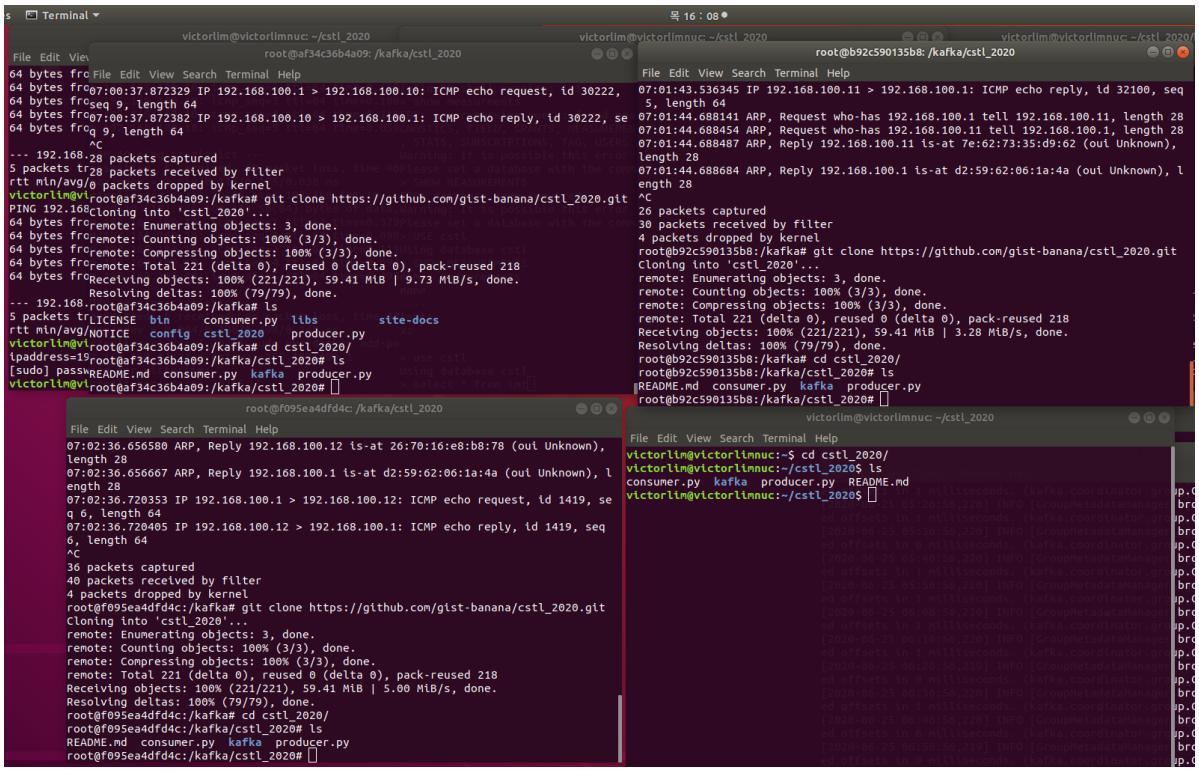


Broker 콘솔(좌), zookeeper 콘솔(우)

다음 단계에서 가이드라인이 요구한 다음 과정은 https://github.com/gist-banana/cstl_2020를 각 컨테이너와 nuc에 설치하는 것인데 이는 producer.py와 consumer.py를 실행하기 위함으로 보인다. 그러나 본 실습에서는 Docker image에 이를 포함시켜 배포하였기 때문에 별도의 설치는 필요하지는 않았지만 진행하여 결과 스크린샷을 첨부한다.

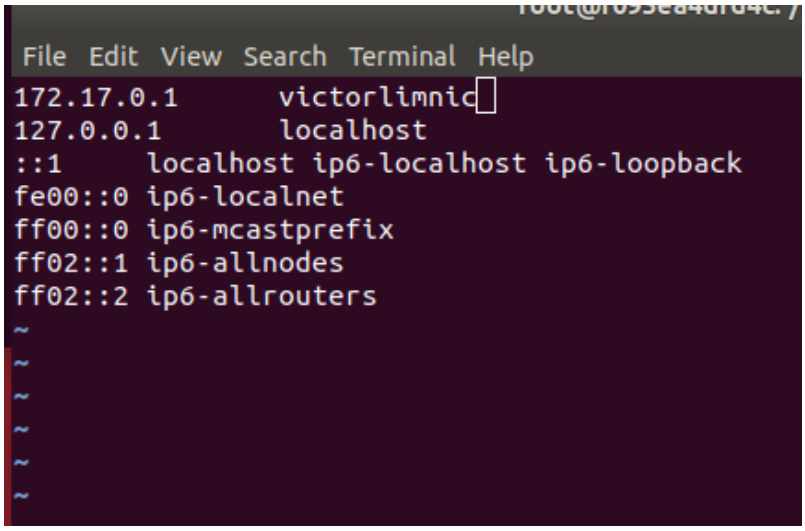
git clone https://github.com/gist-banana/cstl_2020.git

명령어를 이용하였고 git 명령어는 git 패키지를 설정하면 이용가능하다. (이미 도커 이미지에 배포됨)



각 컨테이너와 nuc의 cstl_2020 디렉토리

이제 각 컨테이너들에서 producer.py를 작성하고 실행해서 kafka 클러스터에 연결하는 작업을 진행 하여야 한다. 우선 각 컨테이너에서 nuc의 localhost를 찾을 수 있도록 /etc/hosts에 다음과 같이 추가하였다.



각 컨테이너의 /etc/hosts 파일

각 컨테이너에서 INTERFACE_NAME에는 eno1을 설정하였고 IP_ADDRESS에는 192.168.100.1을 설정하였다. 또한 bootstrap_server에는 kafka 서버의 broker의 주소가 들어가야 하므로 172.17.0.1:9092를 설정하였다. Broker의 ip는 네트워크를 host로 설정하였기 때문에 docker0라는 인터페이스로 broker의 네트워크가 연결되는데 이 ip가 172.17.0.1이였다. 뒤에 9092는 포트 번호로 앞서 broker 컨테이너를 만들 때 설정해준 포트 번호를 써준다.

```
from kafka import KafkaProducer
from kafka.errors import KafkaError
import subprocess as sub

### – START

INTERFACE_NAME = 'eno1' # INPUT NETWORK INTERFCAE THAT YOU WISH TO MONITOR
IP_ADDRESS = '192.168.100.1' # INPUT SOURCE NETWORK IP ADDRESS

# Connect kafka producer here
producer = KafkaProducer(bootstrap_servers=['172.17.0.1:9092'],
                         value_serializer=lambda m: m.encode('utf-8'))
```

```
### - END
topicName = 'cstl'

def transmit_kafka():
    p = sub.Popen(('tcpdump','-i',INTERFACE_NAME,'-l'), stdout=sub.PIPE)

    for line in iter(p.stdout.readline, b''):
        content = str(line)
        if (content.find(IP_ADDRESS)) != -1 and content.find('ICMP') != -1 :
            print(content)
            producer.send(topicName, content)

# transmit_kafka()

transmit_kafka()
```

producer.py의 코드

producer.py를 실행시키기 위해서는 python 관련 패키지가 필요하고 이는 docker image에 포함되어있다. 또한 해당 파이썬파일에서 producer 설정 부분에서 별도의 코드를 추가해줘야 kafka 서버로 알맞은 형식의 데이터를 정상적으로 전송할 수 있게 된다. Producer의 실행은

python3.6 producer.py

명령어를 통해서 이루어지고 바로 tcpdump가 실행되어 eno1을 통해 이동하는 패킷을 모니터링하고 해당 데이터를 kafka 서버로 보내게 된다.

consumer.py을 작성하고 실행시키는 부분이 이 실습에서 가장 어려운 과정 이였다. Consumer는 nuc에서 실행을 시키게 되고 InfluxDB와 연결이 필요하고 기타 여러가지 설정들이 요구되었다. 먼저 bootstrap_server는 producer.py와 마찬가지로 172.17.0.1:9092를 설정하였고 topicname은 cstl을 사용하였다. 35번 라인에 보면 DST_IP, SRC_IP, ICMP_VALUE를 각각 value로부터 parsing 하여

저장하는 코드가 요구되었다. 먼저 value가 NULL인 경우가 발생 가능하므로 세 변수를 초기값으로 널로 지정해준다. 이후 split 함수를 이용해서 value를 space를 기준으로 분리하여 names 배열에 저장하고 이 길이가 7 이상인 경우가 value가 정상적으로 들어온 경우로 판단한다. 이후 세가지의 변수 설정은 space를 기준으로 몇 번째인가를 기준으로 names 배열에서 인덱스 한다. SRC_IP와 ICMP_VALUE의 마지막에는 각각 ':'와 ','가 포함되기 때문에 replace를 이용해서 이를 제거해준다. 이후 consumer.py를 실행시켜보면 오류가 발생하는데 다운받은 파일에 오타가 있는 것으로 생각된다. 아래 첨부한 코드 58번 라인에 measurement에 's'가 붙어있고 그 아래 DST_IP 다음에 ','가 누락되어 발생한 오류로 보여 이를 수정해주었다. 이에 대한 확인은 다음 과정인 데이터를 생성하고 InfluxDB에서 확인하는 순서에서 가능하다.

consumer.py는

python2.7 consumer.py

와 같은 명령어로 수행 가능하다. 여기서 python의 버전 정보가 producer.py의 실행과 다른 이유는 nuc에는 이전 tower lab을 하면서 설치해놓은 python 패키지의 버전이 다르기 때문이다.

완전한 코드는 아래와 같다.

```
# There are total of 2 sections you need to edit.
# Codes you need to edit are designated with
# ### – START
# and
# ### – END

from influxdb import InfluxDBClient
from kafka import KafkaConsumer
import sys
#define information about kafka

### – START
bootstrap_servers = ['172.17.0.1:9092']
topicName = 'cstl'
STUDENT_NAME = "LIMMINTAEK"     # INPUT YOUR STUDENT NAME HERE
```

```
# IF YOU DO NOT INSERT YOUR NAME, YOU WILL GET NO CREDIT


### – END
consumer = KafkaConsumer(topicName, bootstrap_servers = bootstrap_servers)


# connect to InfluxDB


client = InfluxDBClient(host='localhost', port=8086)
client.create_database('cstl')
client.switch_database('cstl')


# database being used : cstl


try :
    for message in consumer:
        value = str(message.value)   # this is where the kafka consumer's value is saved as a string


        # EDIT THE CODES BELOW TO RETRIEVE destination IP address / source IP address / ICMP (request/reply)
        ### – START
DST_IP = ''
        SRC_IP = ''
        ICMP_VALUE = ''


        names = value.split(' ')
        if (len(names)>7) :
            DST_IP = names[2]
            SRC_IP = names[4].replace(':','')
            ICMP_VALUE = names[7].replace(',','')
        # ICMP value means whether the packet is a (request or a reply)
```

```
# You need to have the values printed

print('dst_IP-' + DST_IP)
print('src_ip-' + SRC_IP)
print('icmp_value-' + ICMP_VALUE)

# EDIT THE CODE BELOW TO SAVE VALUES IN THE DATABASEa

### - END
data_to_be_saved = [
        {
            'measurement': 'lmt',
            'tags': {
                'student' : STUDENT_NAME # DO NOT CHANGE THIS LINE
                },
            'fields': {
                'dst_IP' : DST_IP,
                'src_IP' : SRC_IP,
                'value' : ICMP_VALUE
                }
            }
        ]
```

consumer.py의 코드

다음은 이제 구성된 producer-kafka server-consumer를 실제로 동작 시켜보는 단계이다. 앞서 언급하였던 nuc에서 각 컨테이너로 ping을 보내는 것을 tcpdump로 관찰하는 과정과 비슷하다. 가이드라인에서 제시된 3가지 핑 보내기를 수행하고 이를 tcpdump로 관찰한다. 앞선 단계와의 차이는 이제 이 tcpdump의 결과가 consumer를 통해 InfluxDB에 저장된다는 것이다. 아래와 같은 결과는 각 tcpdump에서 확인이 가능하다. 또한 consumer에서 실시간으로 데이터가 쌓이는 것을

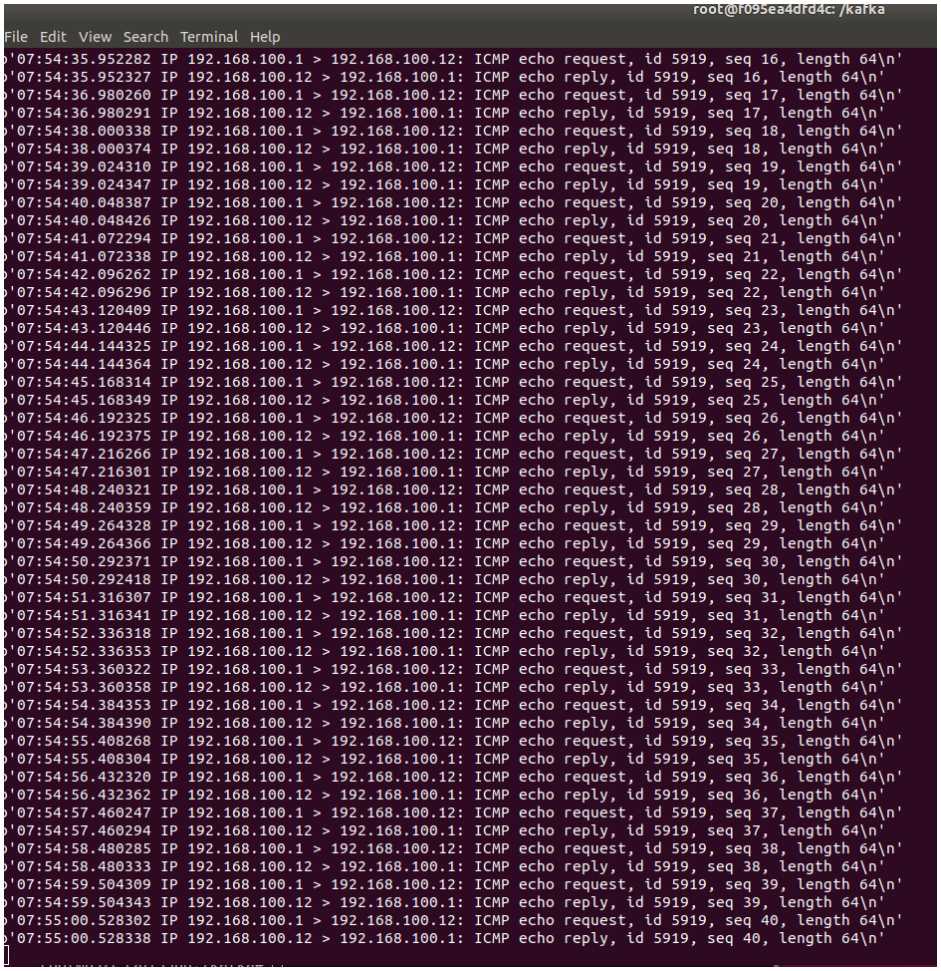
확인 할 수 있었다.

```
07:01:44.688487 ARP, Reply 192.168.100.11 is-at 7e:62:73:35:d9:62 (oui Unknown), length 28
07:01:44.688684 ARP, Reply 192.168.100.1 is-at d2:59:62:06:1a:4a (oui Unknown), length 28
^C
26 packets captured
30 packets received by filter
4 packets dropped by kernel
root@b92c590135b8:/kafka# git clone https://github.com/gist-banana/cstl_2020.git
Cloning into 'cstl_2020'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 221 (delta 0), reused 0 (delta 0), pack-reused 218
Receiving objects: 100% (221/221), 59.41 MiB | 3.28 MiB/s, done.
Resolving deltas: 100% (79/79), done.
root@b92c590135b8:/kafka# cd cstl_2020/
root@b92c590135b8:/kafka/cstl_2020# ls
README.md  consumer.py  kafka  producer.py
root@b92c590135b8:/kafka/cstl_2020# cd ..
root@b92c590135b8:/kafka# rm -r cstl_2020/
root@b92c590135b8:/kafka# vi /etc/hosts
root@b92c590135b8:/kafka# python3.6 producer.py
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eno1, link-type EN10MB (Ethernet), capture size 262144 bytes
b'07:46:00.838998 IP 192.168.100.10 > 192.168.100.11: ICMP echo request, id 858, seq 1, length 64\n'
b'07:46:00.839042 IP 192.168.100.11 > 192.168.100.10: ICMP echo reply, id 858, seq 1, length 64\n'
b'07:46:01.839480 IP 192.168.100.10 > 192.168.100.11: ICMP echo request, id 858, seq 2, length 64\n'
b'07:46:01.839538 IP 192.168.100.11 > 192.168.100.10: ICMP echo reply, id 858, seq 2, length 64\n'
b'07:46:02.864234 IP 192.168.100.10 > 192.168.100.11: ICMP echo request, id 858, seq 3, length 64\n'
b'07:46:02.864272 IP 192.168.100.11 > 192.168.100.10: ICMP echo reply, id 858, seq 3, length 64\n'
b'07:46:03.888195 IP 192.168.100.10 > 192.168.100.11: ICMP echo request, id 858, seq 4, length 64\n'
b'07:46:03.888218 IP 192.168.100.11 > 192.168.100.10: ICMP echo reply, id 858, seq 4, length 64\n'
b'07:46:04.912321 IP 192.168.100.10 > 192.168.100.11: ICMP echo request, id 858, seq 5, length 64\n'
b'07:46:04.912367 IP 192.168.100.11 > 192.168.100.10: ICMP echo reply, id 858, seq 5, length 64\n'
b'07:46:05.936322 IP 192.168.100.10 > 192.168.100.11: ICMP echo request, id 858, seq 6, length 64\n'
b'07:46:05.936355 IP 192.168.100.11 > 192.168.100.10: ICMP echo reply, id 858, seq 6, length 64\n'
b'07:46:06.964207 IP 192.168.100.10 > 192.168.100.11: ICMP echo request, id 858, seq 7, length 64\n'
b'07:46:06.964227 IP 192.168.100.11 > 192.168.100.10: ICMP echo reply, id 858, seq 7, length 64\n'
b'07:46:07.984250 IP 192.168.100.10 > 192.168.100.11: ICMP echo request, id 858, seq 8, length 64\n'
b'07:46:07.984270 IP 192.168.100.11 > 192.168.100.10: ICMP echo reply, id 858, seq 8, length 64\n'
b'07:46:09.008312 IP 192.168.100.10 > 192.168.100.11: ICMP echo request, id 858, seq 9, length 64\n'
b'07:46:09.008344 IP 192.168.100.11 > 192.168.100.10: ICMP echo reply, id 858, seq 9, length 64\n'
b'07:46:10.032316 IP 192.168.100.10 > 192.168.100.11: ICMP echo request, id 858, seq 10, length 64\n'
b'07:46:10.032358 IP 192.168.100.11 > 192.168.100.10: ICMP echo reply, id 858, seq 10, length 64\n'
```

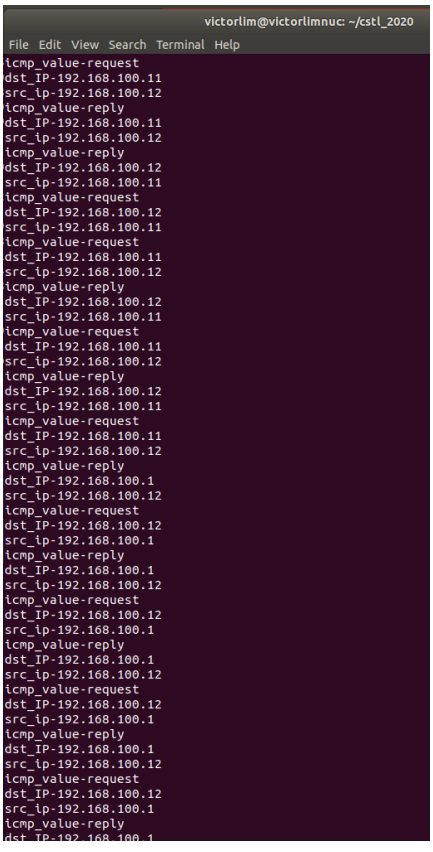
conA → conB로의 10개의 ping을 tcpdump로 관찰한 결과

```
root@b92c590135b8: /kafka
File  Edit  View  Search  Terminal  Help
b'07:49:28.144375 IP 192.168.100.11 > 192.168.100.10: ICMP echo reply, id 860, seq 7, length 64\n'
b'07:49:29.168284 IP 192.168.100.10 > 192.168.100.11: ICMP echo request, id 860, seq 8, length 64\n'
b'07:49:29.168329 IP 192.168.100.11 > 192.168.100.10: ICMP echo reply, id 860, seq 8, length 64\n'
b'07:49:30.192278 IP 192.168.100.10 > 192.168.100.11: ICMP echo request, id 860, seq 9, length 64\n'
b'07:49:30.192312 IP 192.168.100.11 > 192.168.100.10: ICMP echo reply, id 860, seq 9, length 64\n'
b'07:49:31.216278 IP 192.168.100.10 > 192.168.100.11: ICMP echo request, id 860, seq 10, length 64\n'
b'07:49:31.216313 IP 192.168.100.11 > 192.168.100.10: ICMP echo reply, id 860, seq 10, length 64\n'
b'07:49:45.119877 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 1, length 64\n'
b'07:49:45.119934 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 1, length 64\n'
b'07:49:46.128388 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 2, length 64\n'
b'07:49:46.128423 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 2, length 64\n'
b'07:49:47.152206 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 3, length 64\n'
b'07:49:47.152219 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 3, length 64\n'
b'07:49:48.180372 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 4, length 64\n'
b'07:49:48.180406 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 4, length 64\n'
b'07:49:49.200361 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 5, length 64\n'
b'07:49:49.200395 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 5, length 64\n'
b'07:49:50.228339 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 6, length 64\n'
b'07:49:50.228371 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 6, length 64\n'
b'07:49:51.248352 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 7, length 64\n'
b'07:49:51.248395 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 7, length 64\n'
b'07:49:52.276328 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 8, length 64\n'
b'07:49:52.276380 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 8, length 64\n'
b'07:49:53.296350 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 9, length 64\n'
b'07:49:53.296383 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 9, length 64\n'
b'07:49:54.320251 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 10, length 64\n'
b'07:49:54.320266 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 10, length 64\n'
b'07:49:55.344722 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 11, length 64\n'
b'07:49:55.344769 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 11, length 64\n'
b'07:49:56.368276 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 12, length 64\n'
b'07:49:56.368317 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 12, length 64\n'
b'07:49:57.396299 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 13, length 64\n'
b'07:49:57.396340 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 13, length 64\n'
b'07:49:58.416268 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 14, length 64\n'
b'07:49:58.416306 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 14, length 64\n'
b'07:49:59.444249 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 15, length 64\n'
b'07:49:59.444283 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 15, length 64\n'
b'07:50:00.464313 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 16, length 64\n'
b'07:50:00.464360 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 16, length 64\n'
b'07:50:01.488342 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 17, length 64\n'
b'07:50:01.488375 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 17, length 64\n'
b'07:50:02.512379 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 18, length 64\n'
b'07:50:02.512424 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 18, length 64\n'
b'07:50:03.536334 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 19, length 64\n'
b'07:50:03.536369 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 19, length 64\n'
b'07:50:04.560325 IP 192.168.100.12 > 192.168.100.11: ICMP echo request, id 34, seq 20, length 64\n'
b'07:50:04.560360 IP 192.168.100.11 > 192.168.100.12: ICMP echo reply, id 34, seq 20, length 64\n'
```

conC → conB로의 20개의 ping을 tcpdump로 관찰한 결과



nuc → conC로의 40개의 ping을 tcpdump로 관찰한 결과



consumer에서 데이터가 쌓이는 것을 관찰한 결과

이렇게 가져온 데이터들을 InfluxDB에 저장하기 위해서는 우선 measurement를 생성해줘야 한다. 이 작업은

sudo apt-get install influxdb-client

를 실행하여 influxDB의 클라이언트를 설치하고

influx

명령어를 통해서 데이터 베이스에 들어가 몇가지 설정을 해주는 것을 필요로 한다.

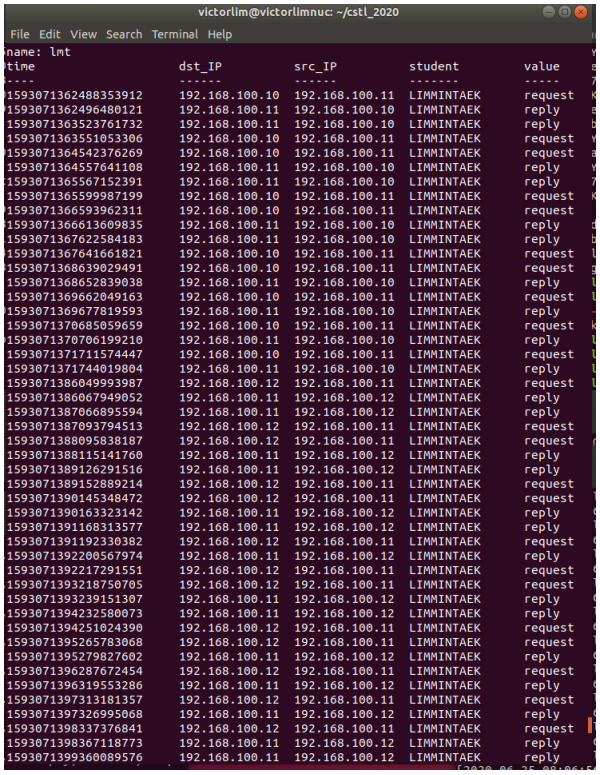
전송된 데이터들이 데이터베이스에 들어왔나 확인하기 위해서는

use cstl

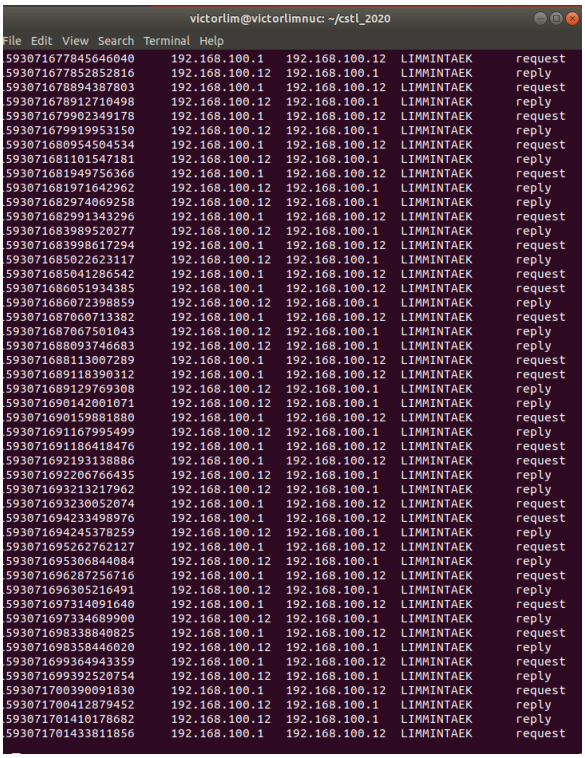
select *from [target]

명령어를 활용했다.

그 결과는 아래와 같다.



influxDB 내의 content(1)



influxDB 내의 content(2)

이러한 과정들을 거쳐서 데이터를 생성하고 kafka 서버에 저장했다가 InfluxBD로 이동시켜 저장하는 전체적인 흐름을 구축할 수 있었다. InfluxBD와 Grafana를 연결해 이를 시각화 하는 작업은 완벽하게 수행하지 못해 아쉬움이 남기도 하지만 그 이전까지의 과정을 구현해보면서 많은 공부를 할 수 있었고 관련 지식들을 얻을 수 있었다. Dockerfile을 작성하여 이미지를 직접 만들고 컨테이너를 작동시키는 원리를 알 수 있었고 컨테이너가 어떠한 방식으로 배포가 되고 이용되는지를 공부하였다. Kafka 서버와 producer, consumer를 연결하는 작업이 가장 어려웠는데 각각의 네트워크 설정과 포트번호를 부여하는 작업이 필요하였고 ip와 포트 번호를 이용하여 서버에 접근하고 데이터를 전송하는 방법을 알 수 있었다. 또한 컨테이너를 생성할 때 네트워크 관련 설정의 종류와 각 경우에 따른 네트워크 연결 방식 역시 알 수 있었다. 실습의 전체적인 흐름은 tower lab과 유사하여 이해가 쉬웠지만 각각의 세부적인 요소들에서 어려움이 많았고 이를 해결하기 위한 공부들이 많이 필요했던 실습 이였다.