Kafka:-

In Big Data, an enormous volume of data is used. Regarding data, we have two main challenges.The first challenge is how to collect a large volume of data and the second challenge is to analyze the collected data. To overcome those challenges, you must need a messaging system.

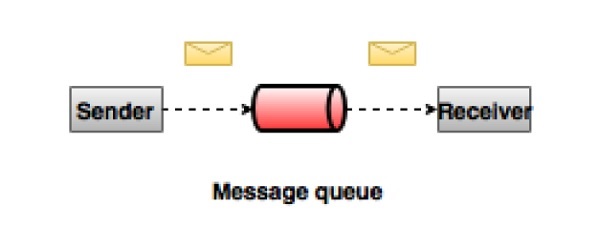
Kafka is designed for distributed high throughput systems. Kafka tends to work very well as a replacement for a more traditional message broker. In comparison to other messaging systems, Kafka has better throughput, built-in partitioning, replication and inherent fault-tolerance, which makes it a good fit for large-scale message processing applications.

## **What is a Messaging System?**

A Messaging System is responsible for transferring data from one application to another, so the applications can focus on data, but not worry about how to share it. Distributed messaging is based on the concept of reliable message queuing. Messages are queued asynchronously between client applications and messaging systems. Two types of messaging patterns are available − one is point to point and the other is publish-subscribe (pub-sub) messaging system. Most of the messaging patterns follow **pub-sub**.

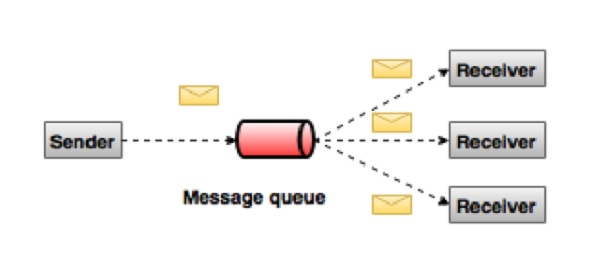
### **Point to Point Messaging System**

In a point-to-point system, messages are persisted in a queue. One or more consumers can consume the messages in the queue, but a particular message can be consumed by a maximum of one consumer only. Once a consumer reads a message in the queue, it disappears from that queue. The typical example of this system is an Order Processing System, where each order will be processed by one Order Processor, but Multiple Order Processors can work as well at the same time. The following diagram depicts the structure.



### **Publish-Subscribe Messaging System**

In the publish-subscribe system, messages are persisted in a topic. Unlike point-to-point systems, consumers can subscribe to one or more topics and consume all the messages in that topic. In the Publish-Subscribe system, message producers are called publishers and message consumers are called subscribers. A real-life example is Dish TV, which publishes different channels like sports, movies, music, etc., and anyone can subscribe to their own set of channels and get them whenever their subscribed channels are available.



## **What is Kafka?**

Apache Kafka is a distributed publish-subscribe messaging system and a robust queue that can handle a high volume of data and enables you to pass messages from one end-point to another. Kafka is suitable for both offline and online message consumption. Kafka messages are persisted on the disk and replicated within the cluster to prevent data loss. Kafka is built on top of the ZooKeeper synchronization service. It integrates very well with Apache Storm and Spark for real-time streaming data analysis.

### **Benefits**

Following are a few benefits of Kafka −

* **Reliability** − Kafka is distributed, partitioned, replicated and fault tolerance.
* **Scalability** − Kafka messaging system scales easily without down time..
* **Durability** − Kafka uses Distributed commit log which means messages persist on disk as fast as possible, hence it is durable..
* **Performance** − Kafka has high throughput for both publishing and subscribing messages. It maintains stable performance even though many TB messages are stored.

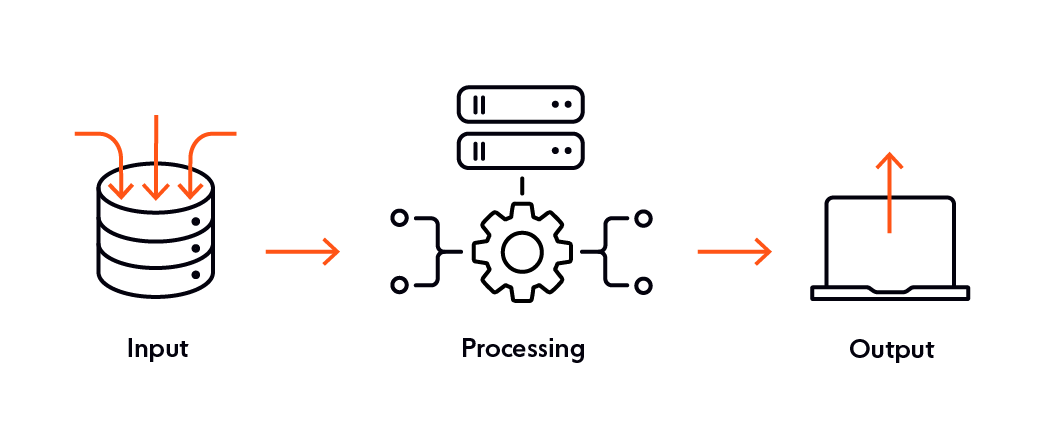
Kafka is very fast and guarantees zero downtime and zero data loss.

## **How Does The Publish-Subscribe Pattern Work?**

Software design patterns are based on building reusable arrangements of modules and their interconnections. These modules are typically classes or objects represented in a UML design diagram. However, when you look at modern architectural patterns, the modules are larger, self-executing processes spread across distributed systems.

To appreciate the advantages of the Pub/Sub pattern, you must start from the basic pattern upon which an information system is built and follow its evolution towards a distributed system.

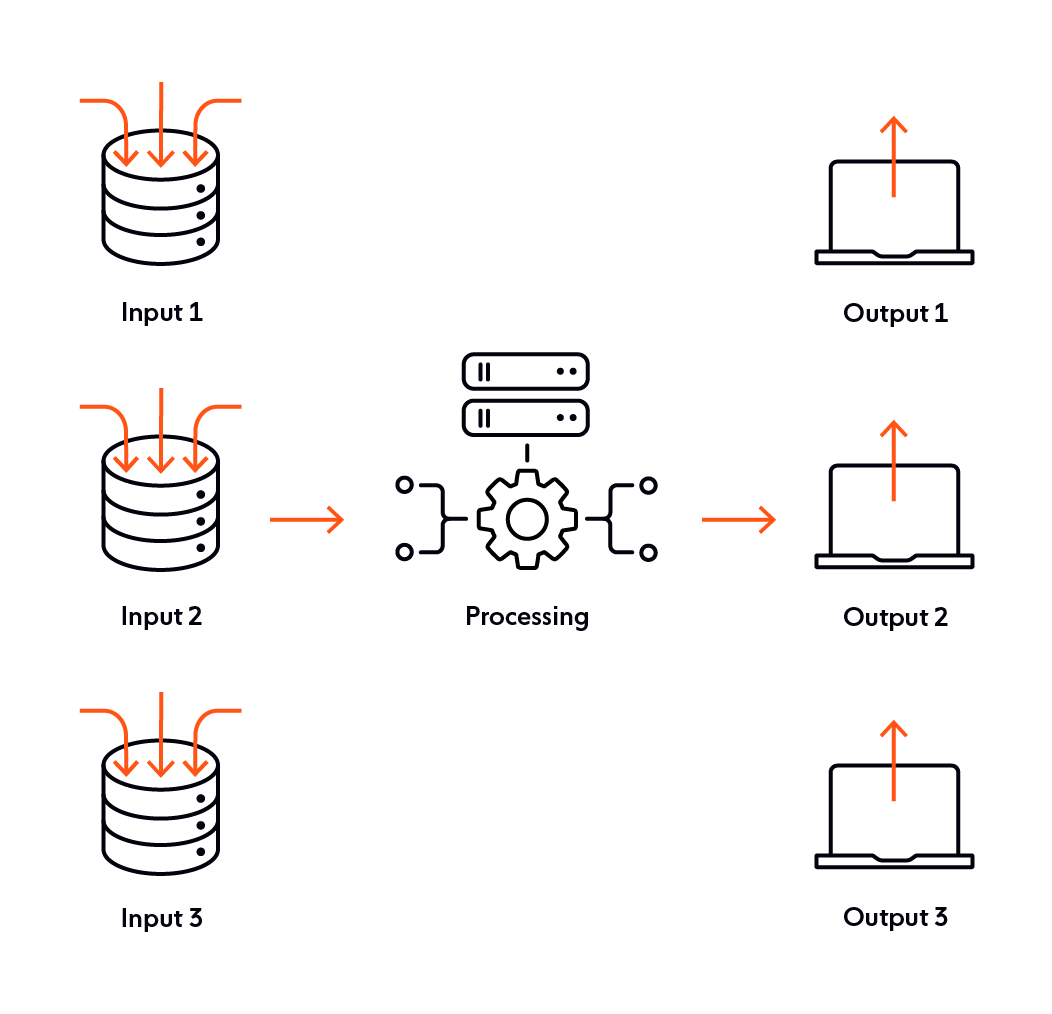
Typically, an information system is an assembly of a generalized set of software modules that follow this simple sequential pattern.



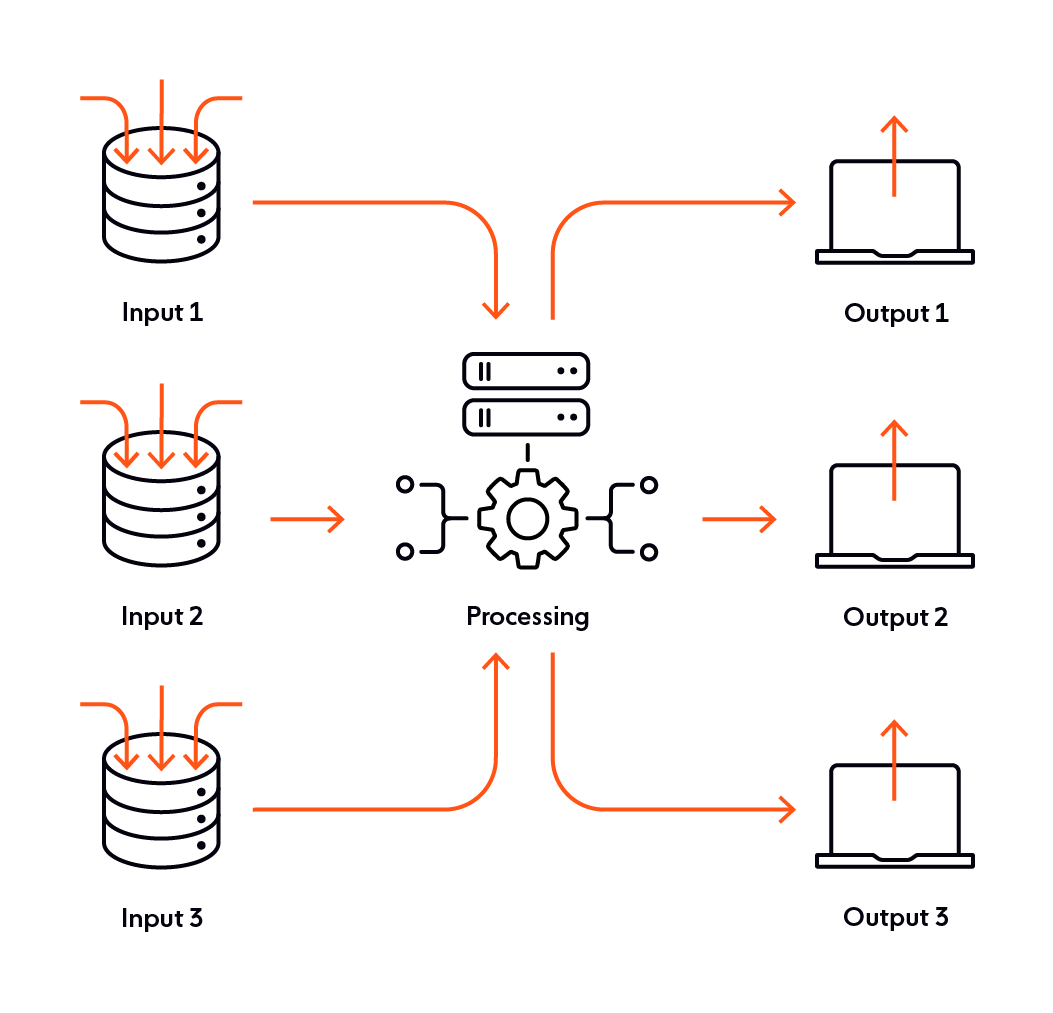
A simple software consisting of three modules.

Think of the illustration above as simple software consisting of three modules. The input module takes the user input and sends the data to the processing module in the form of a message. The processing module processes the data and sends it to the output module as yet another message. The output module displays the data on the user’s screen.

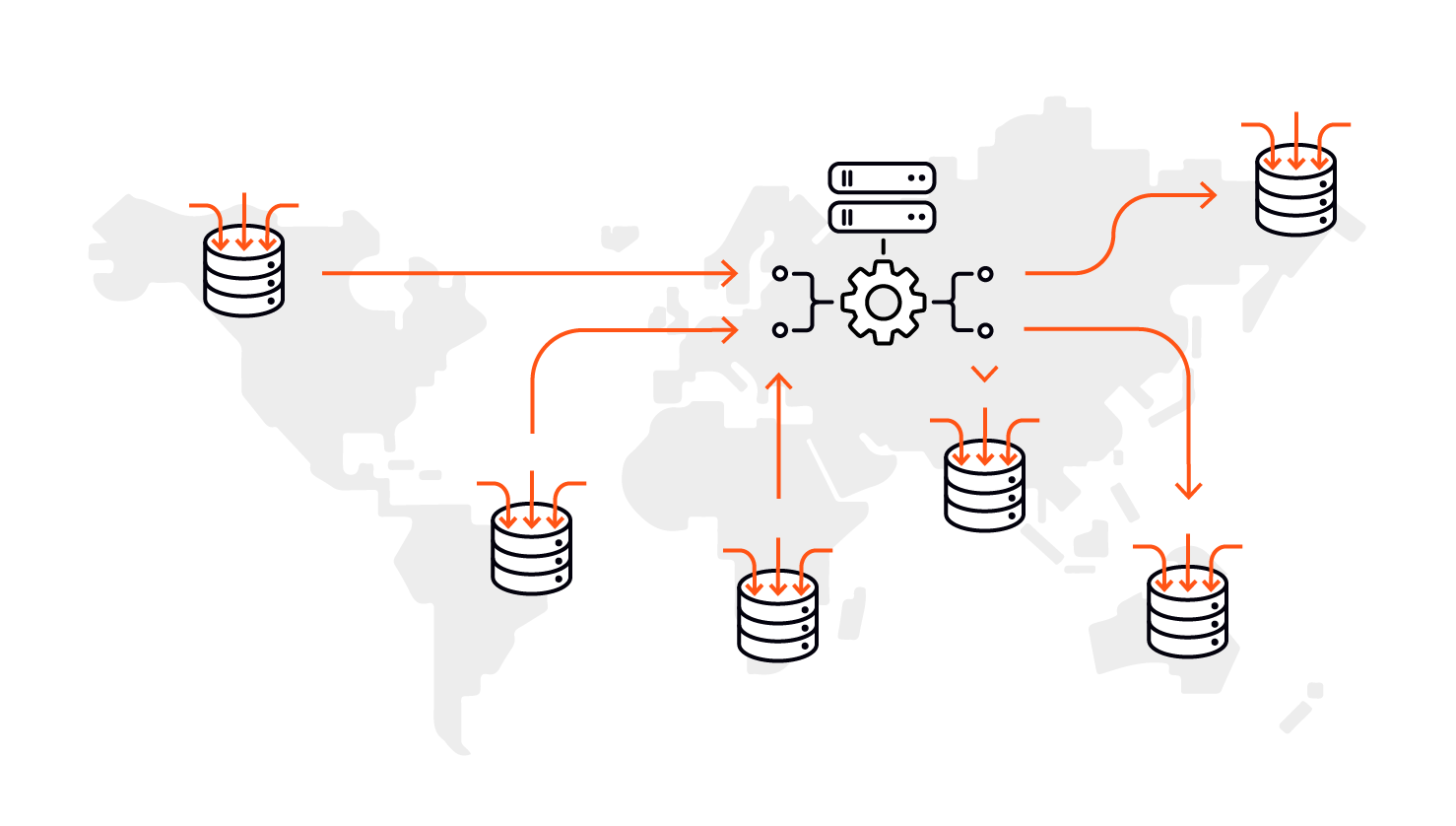
However, the real world is never that simple. At a reasonable scale, the system will need multiple input & output modules for handling concurrent requests.



At this scale, the system faces the problem of routing messages from input modules to their respective output modules. To solve this problem, the input and output modules will need an addressing mechanism. The processing module will process the messages and route them to the correct recipient based on an address. All three modules collaborate in solving the problem of routing.



At Internet scale, the system will handle thousands of concurrent connections. The system will receive messages from and send messages to users all over the world. It needs to also be capable of handling high volume and global geographical spread of users.



However, at such a massive scale, the system modules will not work as expected.

* The processing module can’t handle the load. Because of the high volume and geographical spread, the load needs to be distributed between multiple processing modules.
* At this scale, the dynamics of input and output change. Pre-defined addressing between the modules becomes a huge overhead.

You can solve the first problem by introducing multiple processing modules. This has the effect of splitting the system horizontally. However, this increases routing complexity. The input modules must now route the messages to the correct processing module.

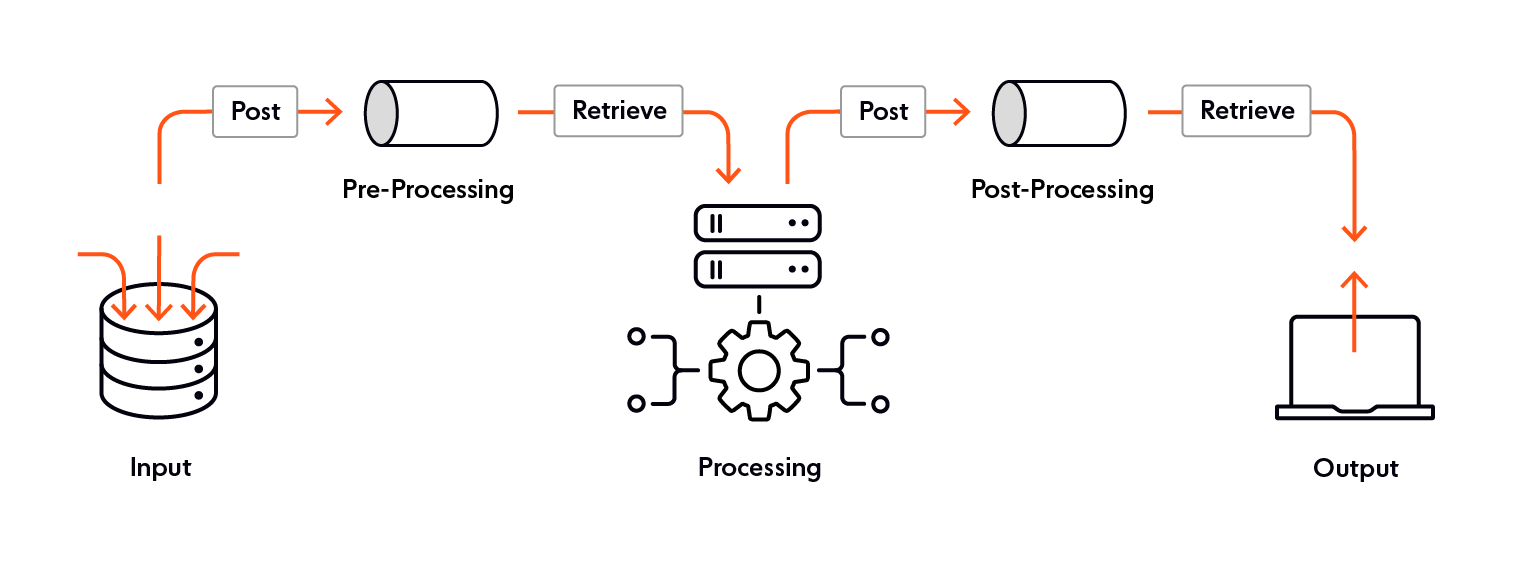
Attaching module specific routing metadata to messages becomes a bottleneck at internet-scale. Under such circumstances, the design of a message passing from one module to the next begs a radical rethink.

## **Enter Pub/Sub**

Having to program the modules to maintain a shared knowledge of addresses for other modules is burdensome for developers. The complexity of this dependency will increase with scale and it will eventually break the system.

The best way to ease this burden is to minimize the shared knowledge of addresses. To achieve this, you can tweak the design of the modules such that they perform their tasks and then interact through a common forum.

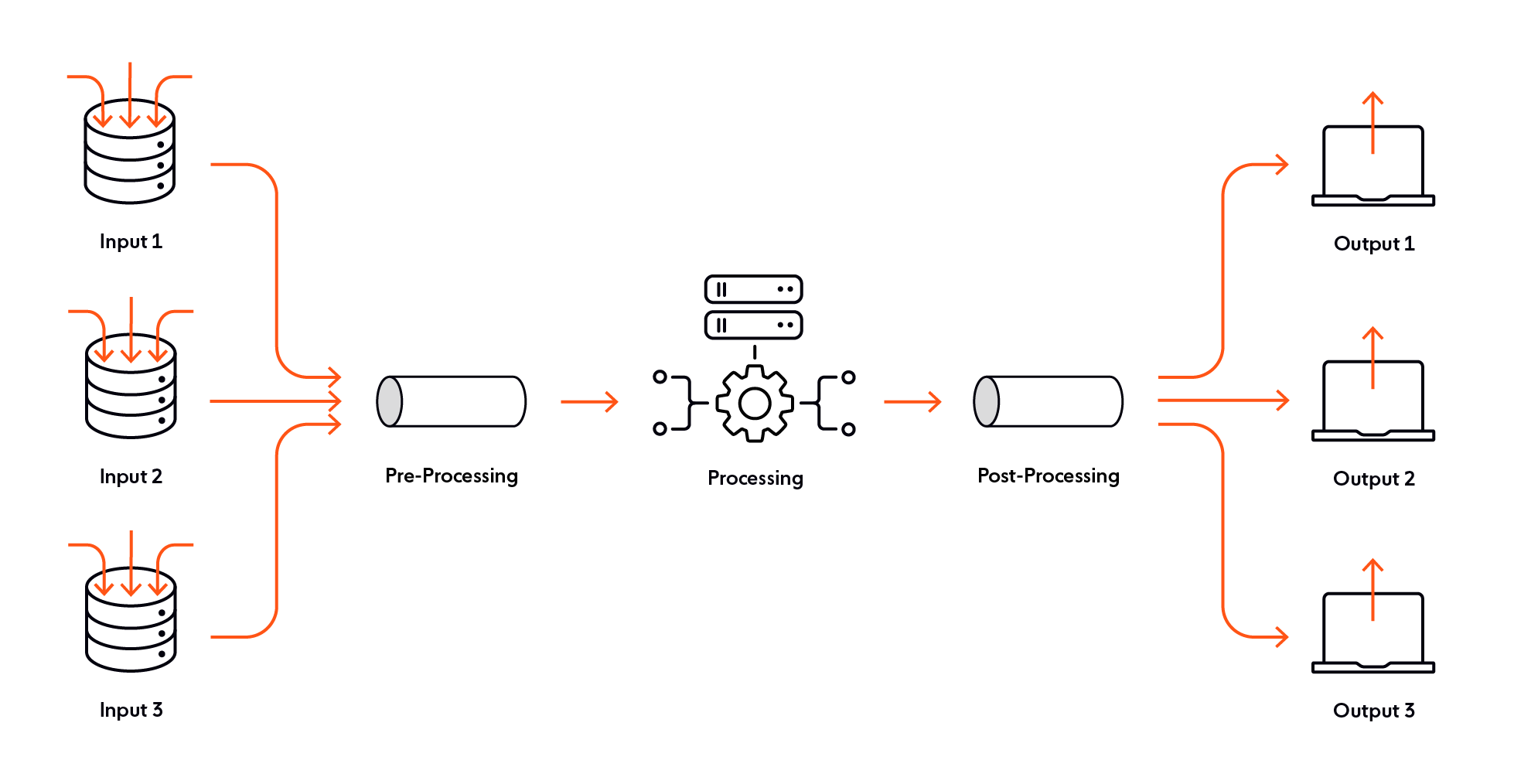
You can think of this common forum as a channel or a data pipe. The modules can post their messages to it or retrieve the messages posted by other modules from it.



With this architecture, a developer can program the modules to have isolated and well-defined responsibilities, so the modules no longer need to maintain shared knowledge on the whereabouts of other modules. The input modules only accept user input, processing modules only process the data, and the output modules only display the output.

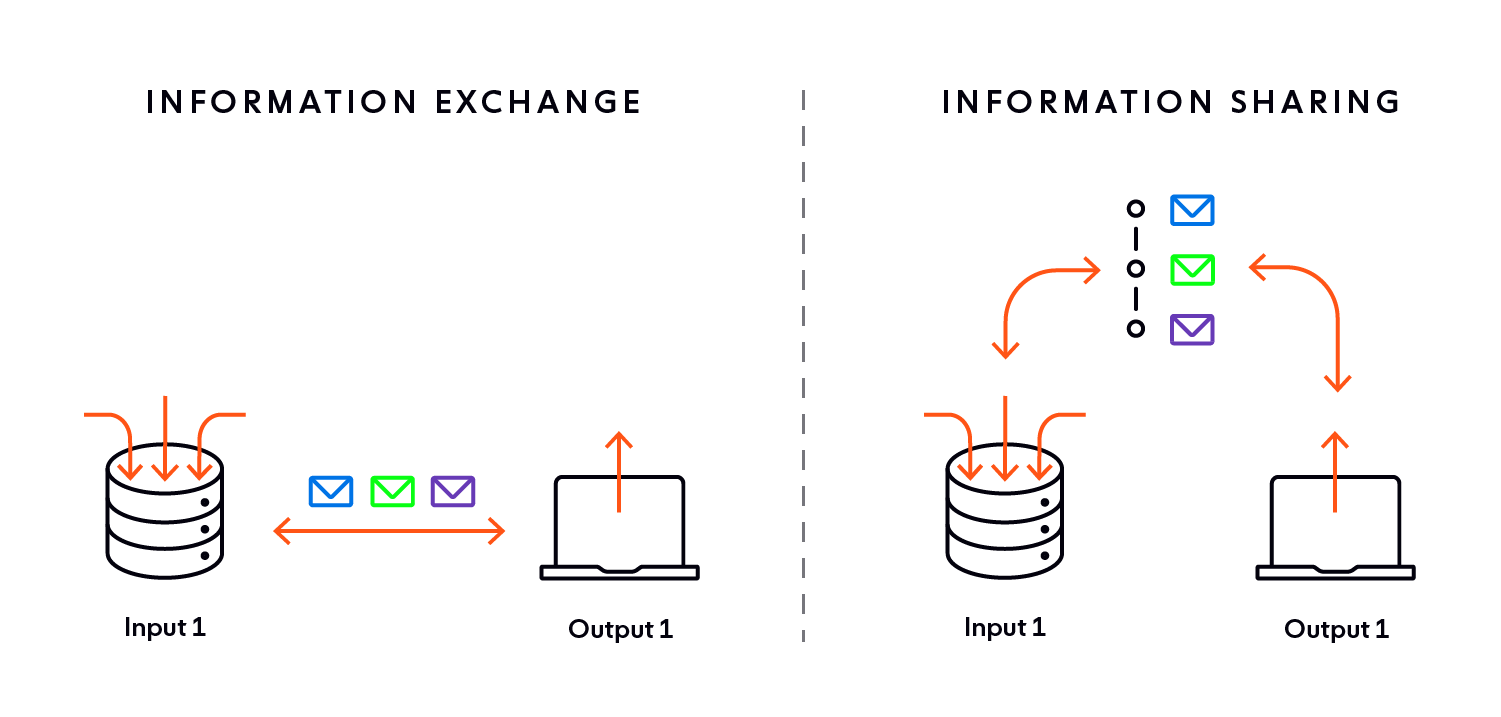
The only additional thing the modules need to know is the channel for posting and retrieving messages. The input module will gather the user input and post the message in the preprocessing channel. The processing module will pick the messages from this channel, process it and post it to the post-processing channel. Finally, the output module will collect the message from the post-processing channel and display it on the users’ screen.

The same pattern can be followed at any scale.



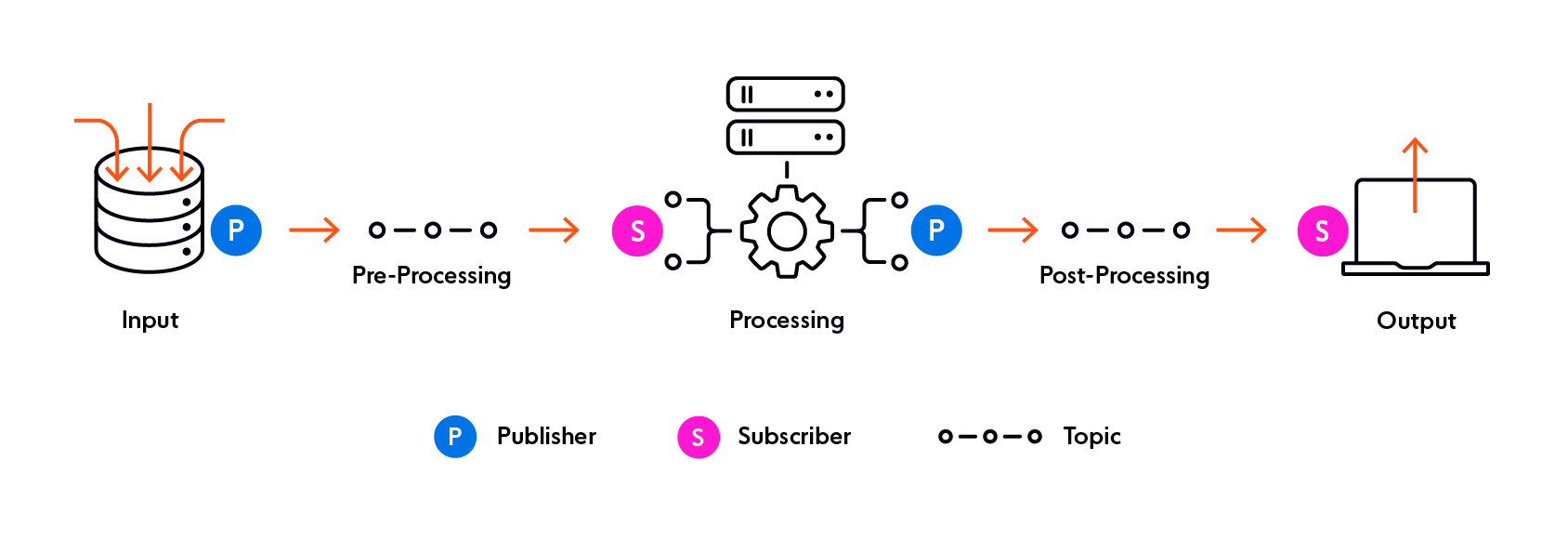
But what about the original routing problem? How do the input modules know about their corresponding output modules or vice versa?

The short answer is, they don’t need to know. The problem of a one-to-one input-output address mapping is now transformed to posting to a channel and retrieving from a channel. Therefore, you are now witnessing a change in the paradigm of communication from point-to-point information exchange to information sharing on a common forum.



This approach of sharing information forms the fundamental repeat unit of the Pub/Sub pattern.

To recap on the initial overview, in the Pub/Sub parlance, you call the module that posts messages a publisher. You call the module that retrieves messages from a subscriber. The publisher publishes messages on a topic. The subscriber subscribes to the topic to receive messages. You can now assemble the generalized information system as shown below.

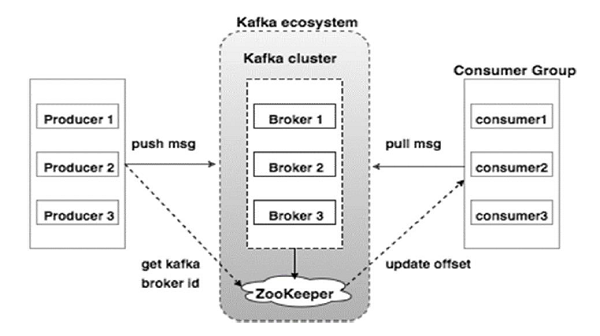


The input module publishes the messages on the pre-processing topic. The processing module also subscribes to this topic to receive all messages published by the input module. The processing module further acts as the publisher of processed messages on the post-processing topic. The output module subscribes to this topic.

The topics are like virtual pathways. You can create and destroy them on the fly. This makes the management and administration of topics a separate responsibility, abstracted from the modules. As a result, a developer is not faced with additional complexity in programming the modules, even at scale.

The responsibility of managing the topics is now entirely shifted to the message broker. The message broker is an independent component and has its own implementation for the administration of published messages and their delivery to the subscribers. As Pub/sub is a design pattern, it does not address these implementation details.

## Kafka - Architecture



**Broker**

Kafka cluster typically consists of multiple brokers to maintain load balance. Kafka brokers are stateless, so they use ZooKeeper for maintaining their cluster state. One Kafka broker instance can handle hundreds of thousands of reads and writes per second and each bro-ker can handle TB of messages without performance impact. Kafka broker leader election can be done by ZooKeeper.

**ZooKeeper**

ZooKeeper is used for managing and coordinating Kafka brokers. ZooKeeper service is mainly used to notify producer and consumer about the presence of any new broker in the Kafka system or failure of the broker in the Kafka system. As per the notification received by the Zookeeper regarding presence or failure of the broker then producer and consumer takes decision and starts coordinating their task with some other broker.

**Producers**

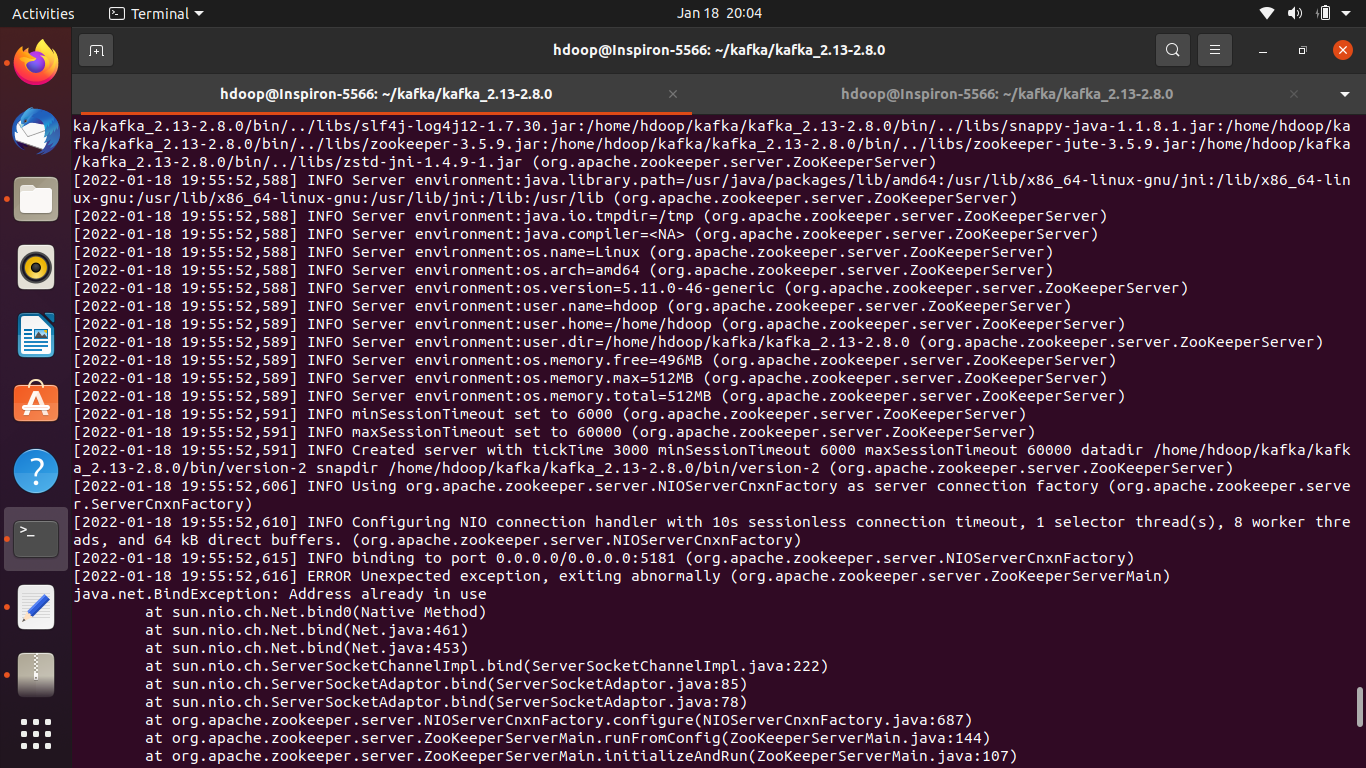
Producers push data to brokers. When the new broker is started, all the producers search it and automatically send a message to that new broker. Kafka producer doesn’t wait for acknowledgements from the broker and sends messages as fast as the broker can handle.

**Consumers**

Since Kafka brokers are stateless, which means that the consumer has to maintain how many messages have been consumed by using partition offset. If the consumer acknowledges a particular message offset, it implies that the consumer has consumed all prior messages. The consumer issues an asynchronous pull request to the broker to have a buffer of bytes ready to consume. The consumers can rewind or skip to any point in a partition simply by supplying an offset value. Consumer offset value is notified by ZooKeeper.

## Kafka installation

hdoop@Inspiron-5566:~/kafka/kafka\_2.13-2.8.0$ **bin/zookeeper-server-start.sh config/zookeeper.properties**



ERROR:

2022-01-18 19:27:58,237] INFO Configuring NIO connection handler with 10s sessionless connection timeout, 1 selector thread(s), 8 worker threads, and 64 kB direct buffers. (org.apache.zookeeper.server.NIOServerCnxnFactory)

[2022-01-18 19:27:58,242] INFO binding to port 0.0.0.0/0.0.0.0:5181 (org.apache.zookeeper.server.NIOServerCnxnFactory)

[2022-01-18 19:27:58,243] ERROR Unexpected exception, exiting abnormally (org.apache.zookeeper.server.ZooKeeperServerMain)

**java.net.BindException: Address already in use**

at sun.nio.ch.Net.bind0(Native Method)

at sun.nio.ch.Net.bind(Net.java:461)

at sun.nio.ch.Net.bind(Net.java:453)

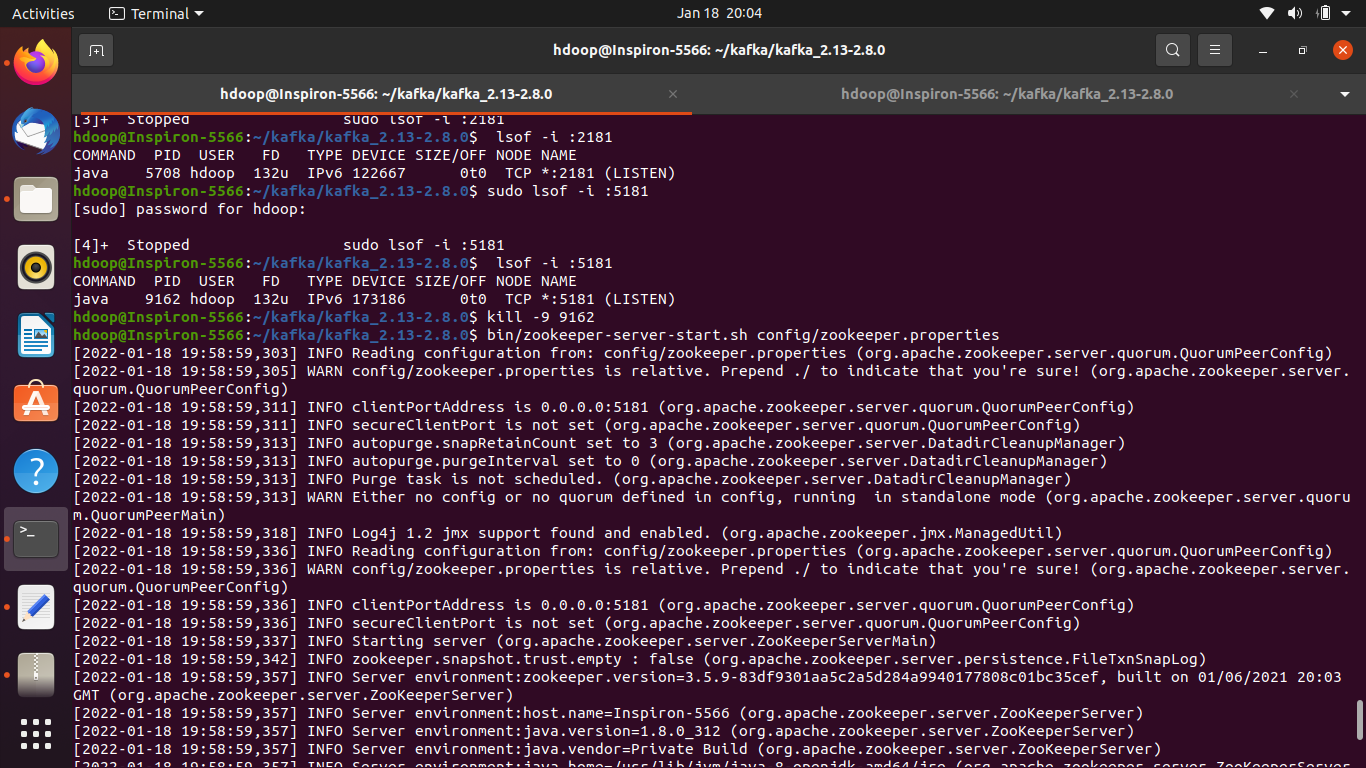
at sun.nio.ch.ServerSocketChannelImpl.bind(ServerSocketChannelImpl.java:222)

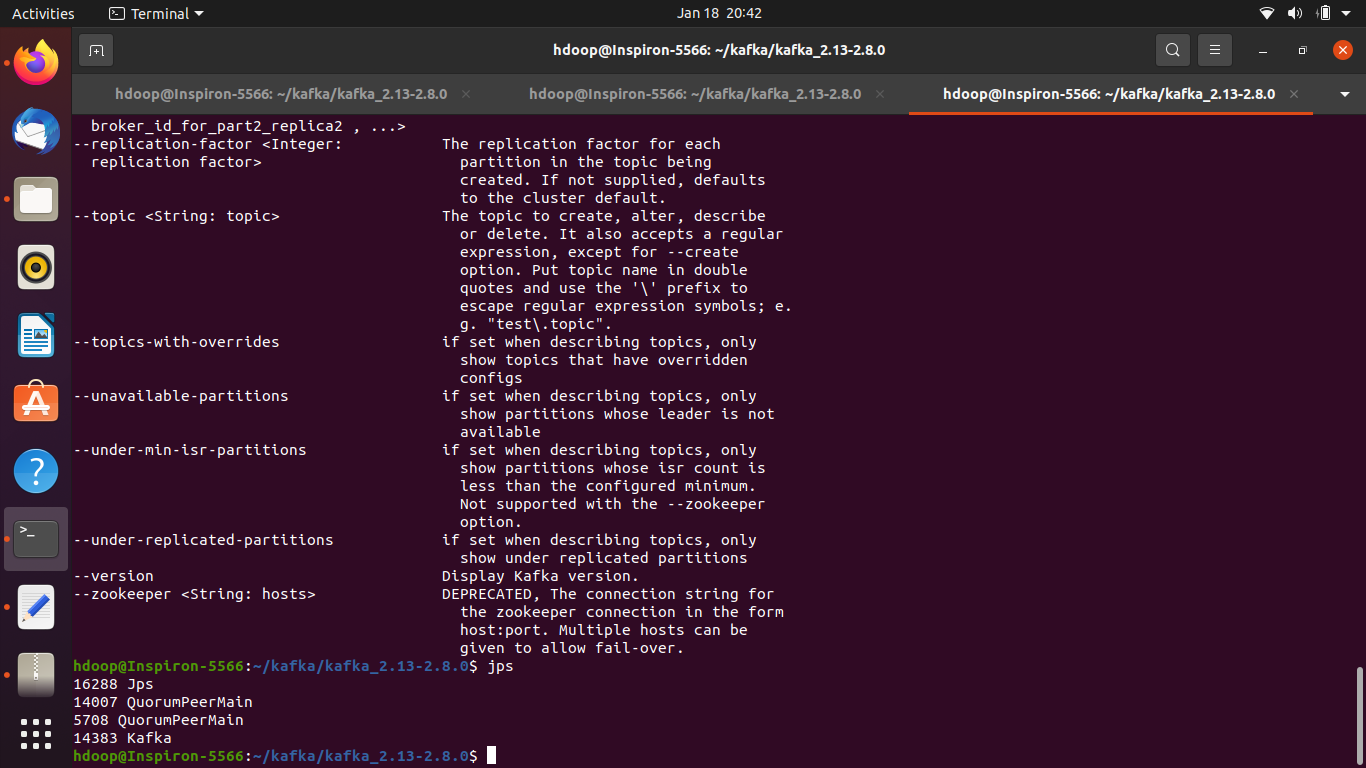
at sun.nio.ch.ServerSocketAdaptor.bind(ServerSocketAdaptor.java:85)

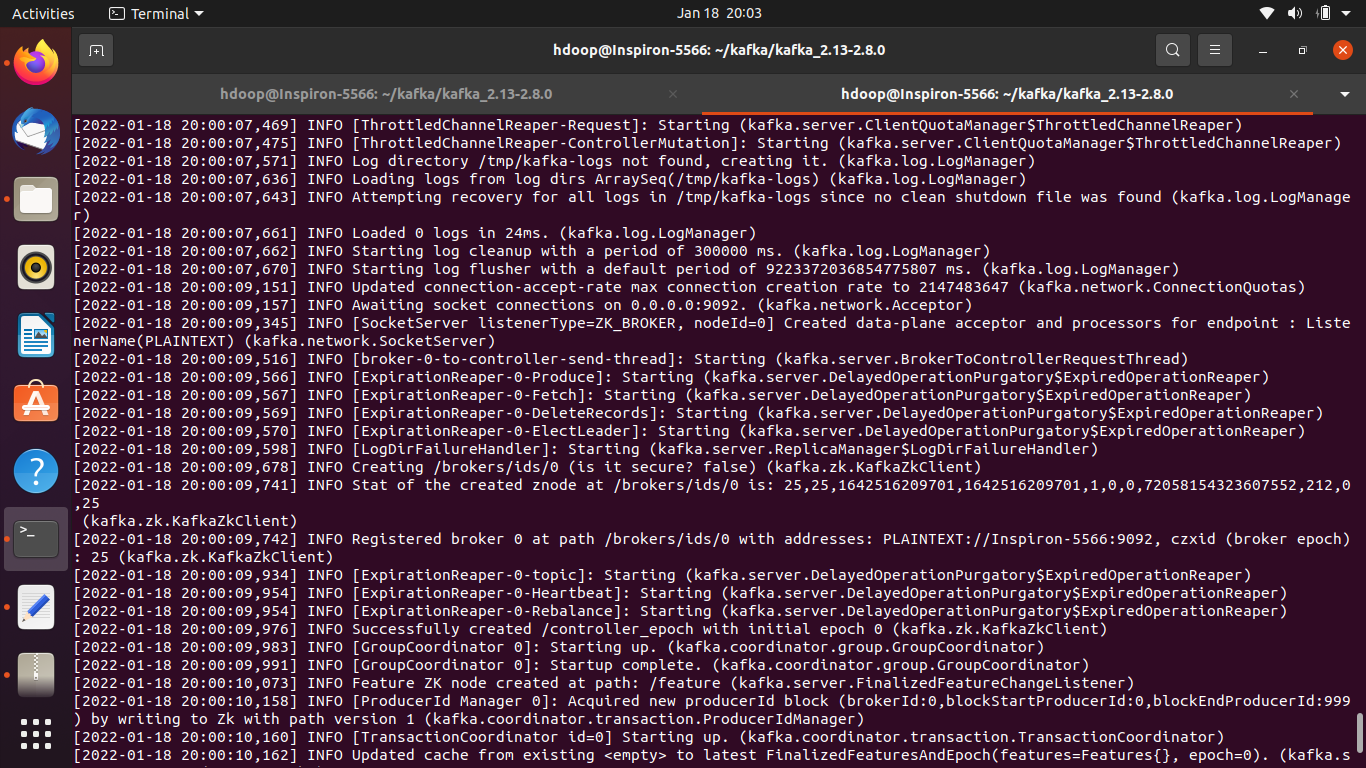
at sun.nio.ch.ServerSocketAdaptor.bind(ServerSocketAdaptor.java:78)

SOLUTION:

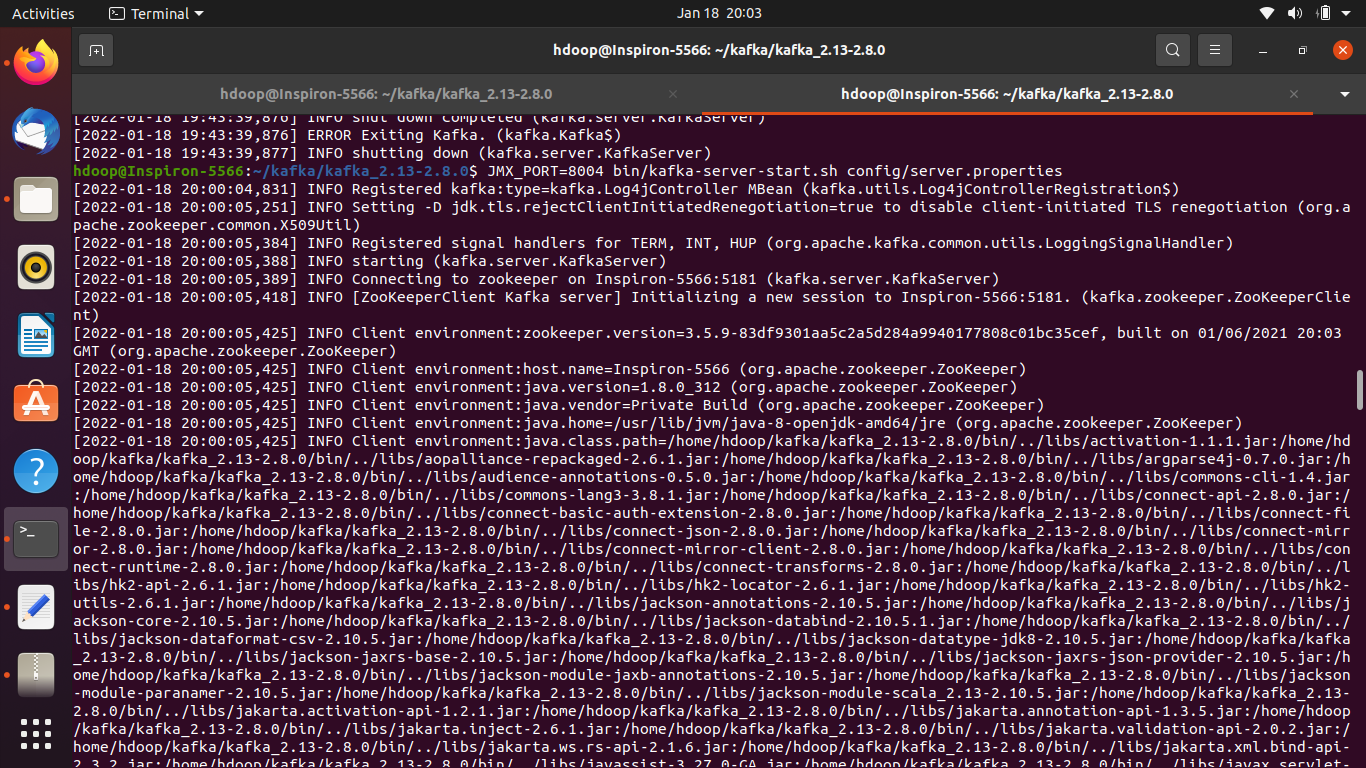
https://stackoverflow.com/questions/43293870/i-cant-run-zookeeper

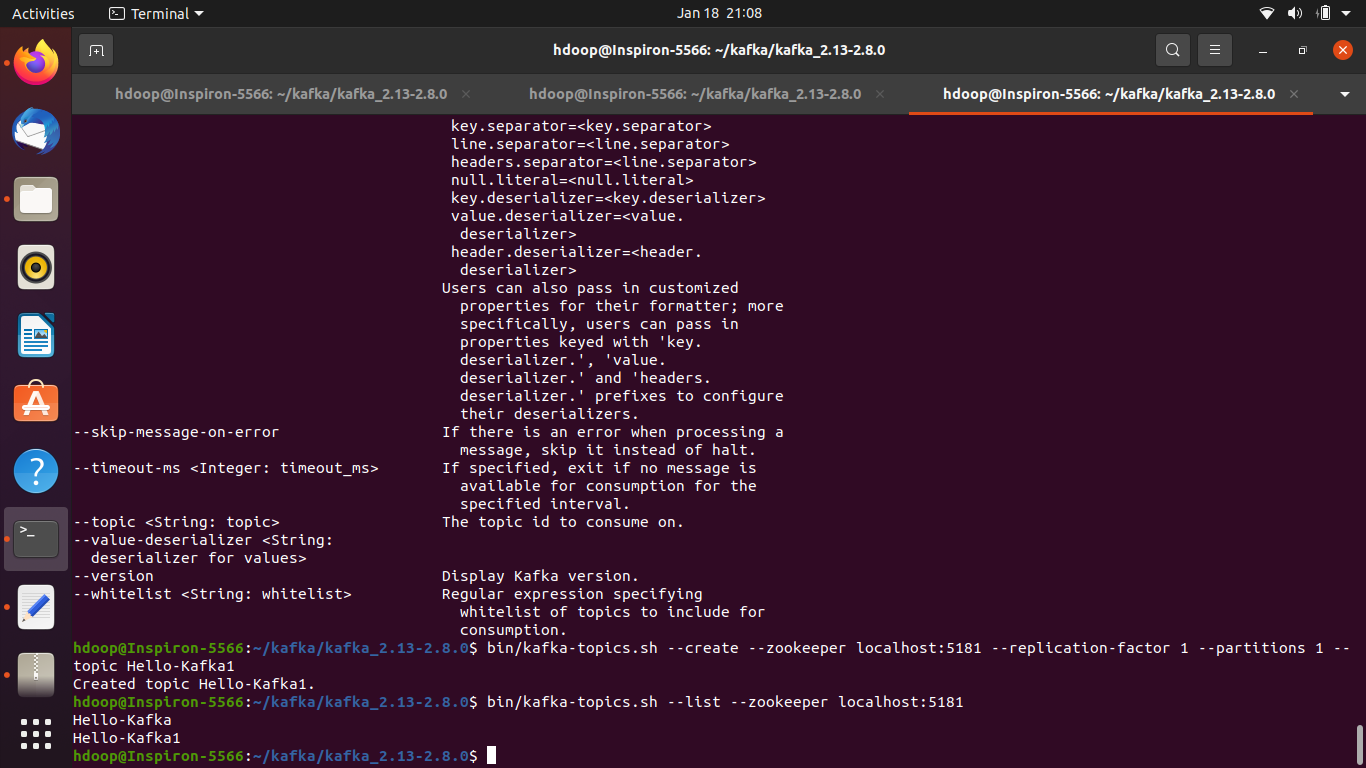
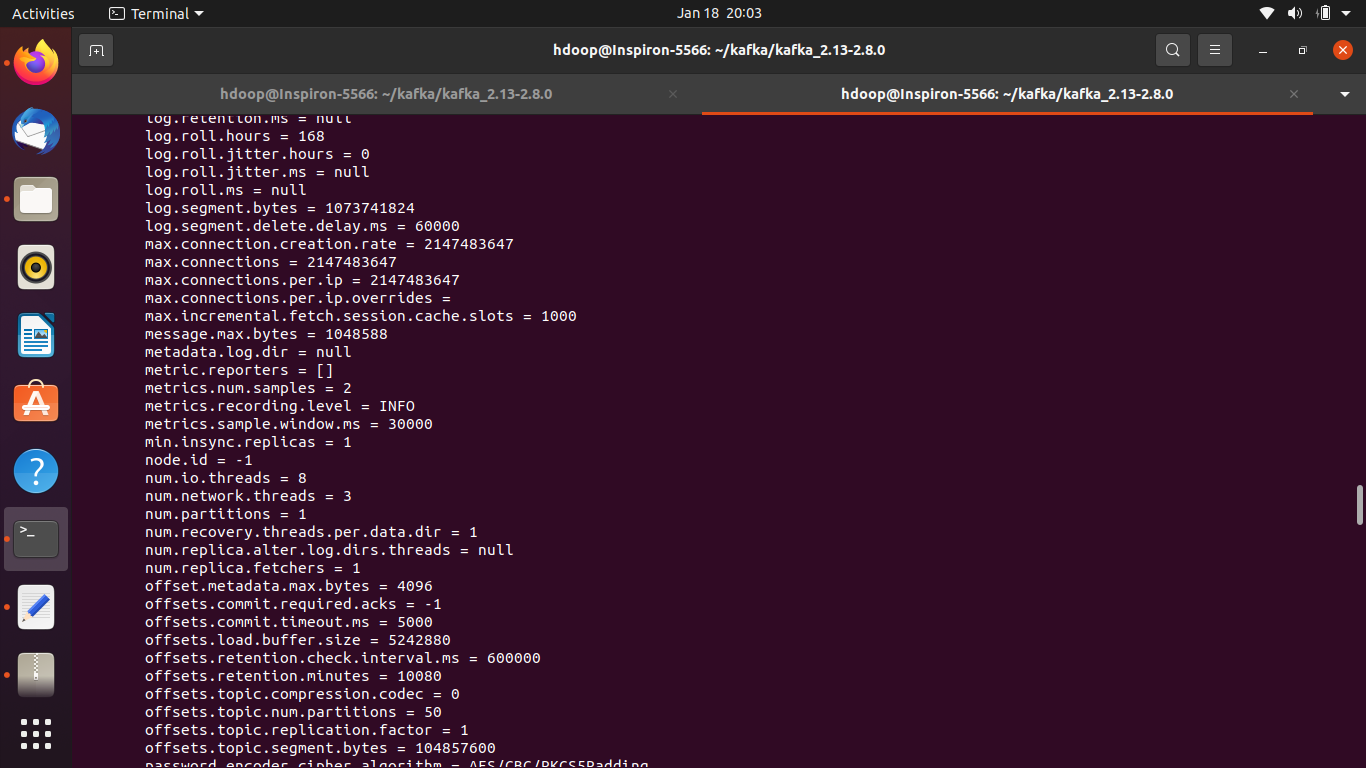






/kafka/kafka\_2.13-2.8.0$ **JMX\_PORT=8004 bin/kafka-server-start.sh config/server.properties**





**Starting zookeper**

hdoop@Inspiron-5566:~/kafka/kafka\_2.13-2.8.0$ bin/zookeeper-server-start.sh config/zookeeper.properties

**Starting kafka**

hdoop@Inspiron-5566:~/kafka/kafka\_2.13-2.8.0$ JMX\_PORT=8004 bin/kafka-server-start.sh config/server.properties

### Create topic

bin/kafka-topics.sh --create --zookeeper localhost:5181 --replication-factor 1--partitions 1 --topic topic-name

bin/kafka-topics.sh --create --zookeeper localhost:5181 --replication-factor 1 --partitions 1 --topic Hello-Kafka

bin/kafka-topics.sh --list --zookeeper localhost:5181

### Producer producing messages on terminal

hdoop@Inspiron-5566:~/kafka/kafka\_2.13-2.8.0$ kafka-console-producer.sh --broker-list localhost:9092 --topic Hello-Kafka

>my first message

>will display here

>

### Consumer reading messages on terminal

hdoop@Inspiron-5566:~/kafka/kafka\_2.13-2.8.0$ kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic Hello-Kafka --from-beginning

my message is here

my first message is here

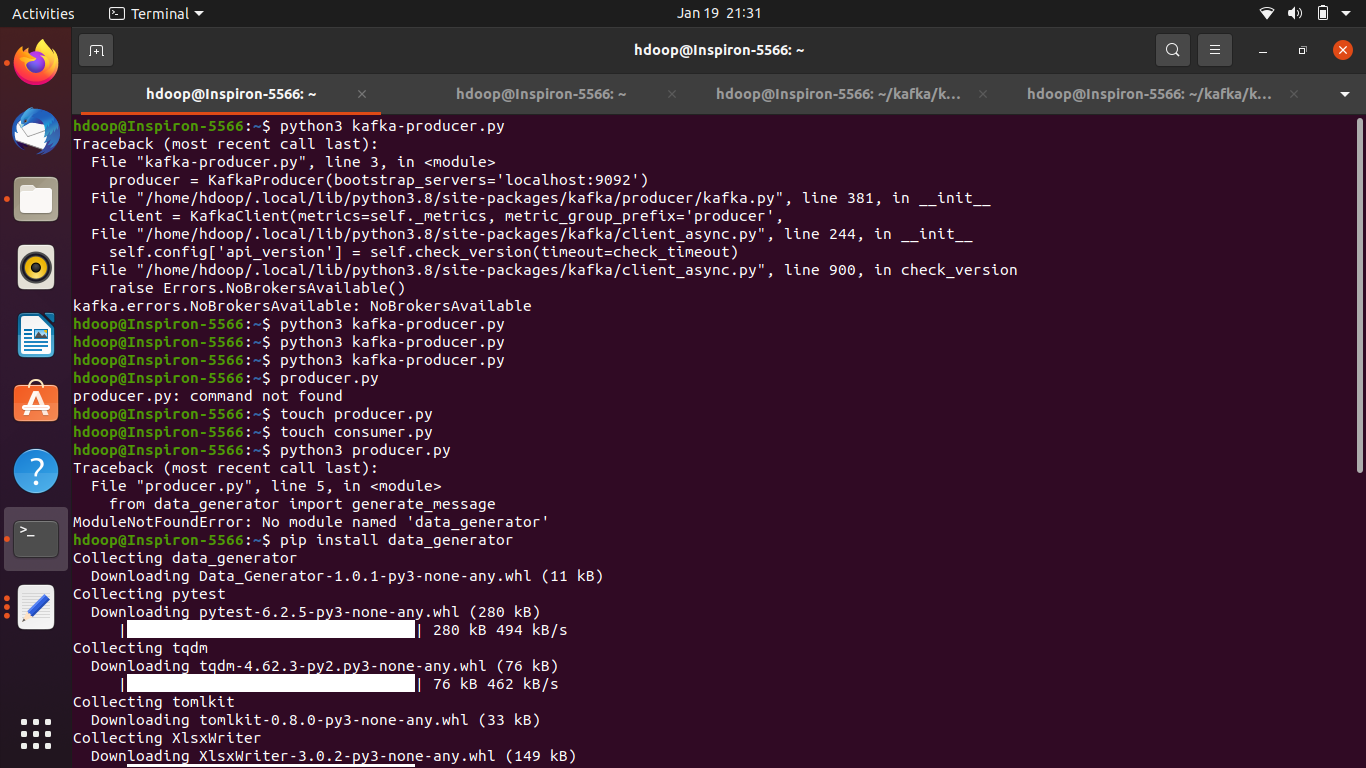
hlo

heloo

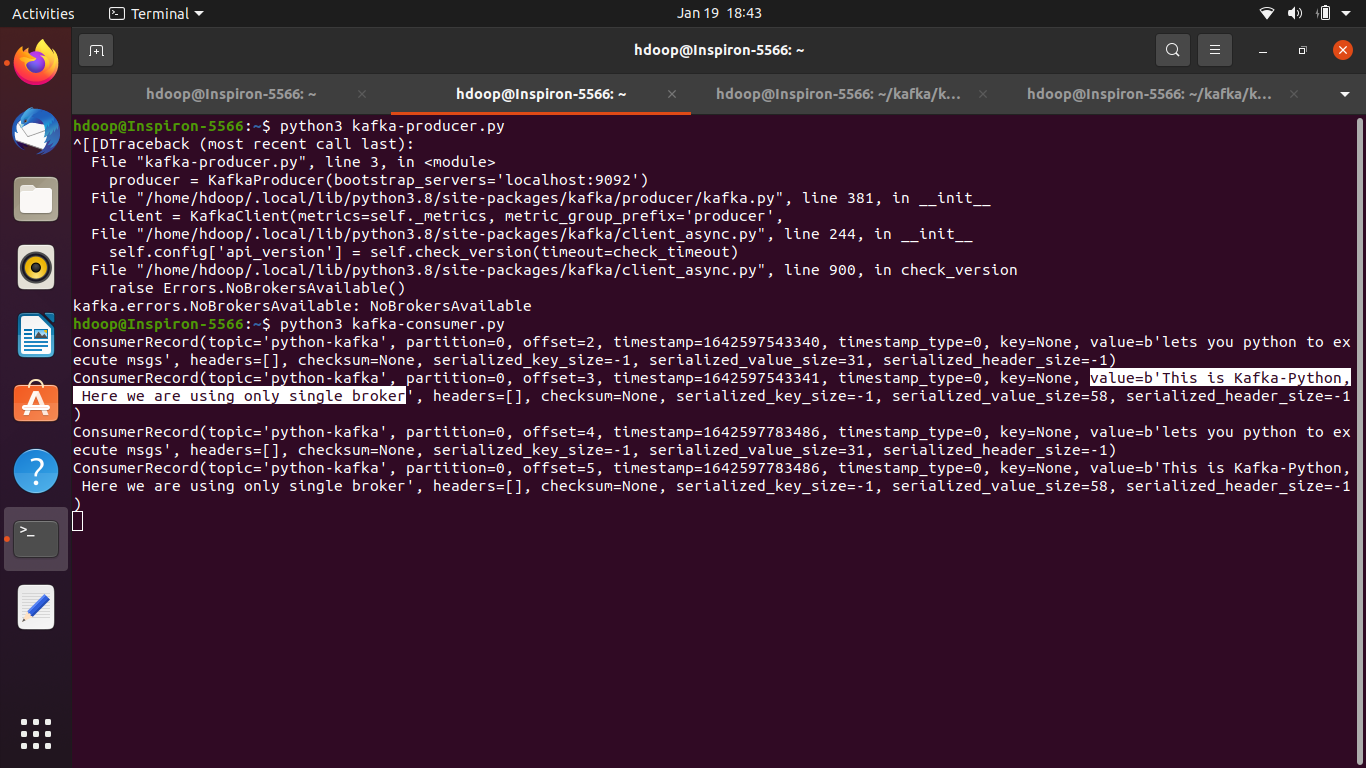
my first message

will display here

$ python3 kafka-producer.py



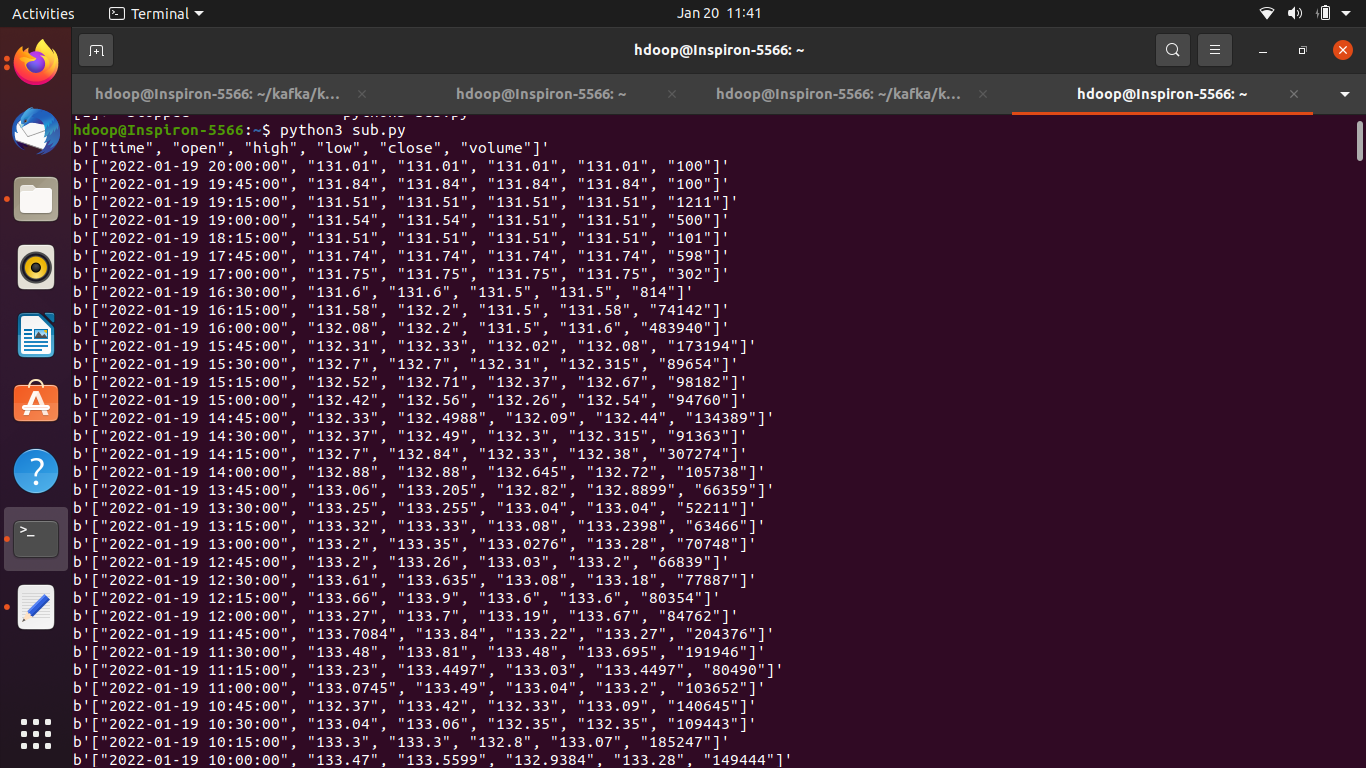
$ python3 kafka-consumer.py



# Taking realtime data using python code in kafka

$ python3 pub.py

$ python3 sub.py



**hdoop@Inspiron-5566:~$ python3 stock\_data\_hdfscmd.py**

**Output:-** Running system command: ['hdfs', 'dfs', '-mkdir', '/Kafka\_Stock\_Data']

(0, b'', b'2022-01-20 12:22:45,489 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable\n')

Running system command: ['hdfs', 'dfs', '-copyFromLocal', '/home/hdoop/stock\_data.txt', '/Kafka\_Stock\_Data']

(0, b'', b'2022-01-20 12:22:51,148 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable\n')

