# Realtime Stockdata Messaging System

Leiquan Pan, Jiangnan Liu
lpan22@wustl.edu, liu433@wustl.edu

**Abstract**

*Currently, there exists different kinds of real-time messaging system such as NSQ and kafka. Also, there are different kinds of stock data messaging system. However, some of the stock trading platforms cannot respond to the market immediately as soon as there are price fluctuations on the market. They do not adequately solve the latency challenge and the users have limit choices over what kind of stock data they would like to view. To overcome this limitation, we have developed our software called RSMS that integrates a stock database from Quandl, a data processing platform Apache Flink and the open source real-time messaging middle ware NSQ. This software features real-time feedback from the stock market data and requires little hand tuning from users' side. This paper presents the design, implementation, and evaluation of the RSMS.*

## 1 Introduction

Nowadays, more and more people start to choose stock as an investment way and they need to get new and fresh stock data once there are up and down in the market to make a choice on whether buy or sell. If the stock price goes high in the market, users may make the choice to purchase more before it rise even higher or to sell it immediately before the price drop. Thus, users need to be notify once there is fluctuations on the market and latency is a crucial factor in designing the system. once th Latency is very important in the processing and delivery of valuable stock data information to the subscribers of specific stocks. However, current stock applications only focus on providing notification when a specific stock rise or drop and the general trend of a group of stock is not paying enough attention. The applications cannot ensure react and report to users once there are fluctuations on the market. Sometimes, when the users receive a notification and decide to make a purchase or sell request, the actual stock price has already changed and the price reported to the user has been expired. Besides, there are thousands of stocks in the market across the country and the stream of stock need to be filtered and processed before the delivery to the users. We are motivated by these current problems and would like to introduce a need stock processing and delivery platform called RSMS.

Generally, our system can be divided into two separate components. The first part is stock information filtering and processing. RSMS use Flink to filter and process stock data stream and label each record. After processing, Flink sink each stock record to the specific topic determined by label. The second part of our system is stock information delivery. The stock information filtering and processing platform stream data into different topics and publish them to the middleware broker NSQD. After the middleware broker receive request from subscriber to specific topics, it delivers all the stock information and send them directly to the subscribers. We also implement a topology information manager nsqlookupd to which users can query and it will find the NSQD producers for the specific topic.

In general, the objective of our RSMS system is to achieve scalability, low latency, independence of processing middleware and reliability. The performance of the system will be evaluated based on the above criteria and reducing latency is our main goal in the process of designing and implementing the platform.

## 2 Background & Related Work

The Real-time requirements of both processing platform and messaging service contribute to some open source projects such as Apache Flink, kinesis and Storm in data processing and Kafka, NSQ and MQTT in data forwarding. There exits many implementations of processing and messaging services in real applications. In NYC taxi tracking project, which is a real-time taxi data process and analysis software created by AWS engineers, kinesis is applied to create data stream and forward raw data, Flink was used to effectively process data. Their results shows high accuracy, promising stability and strong processing. However, in our case, we focused mainly on real time performance and distributed Features. We agree to apply Apache Flink as our processing middleware, since its real time performance and fault tolerant mechanism. And we consider NSQ as our message service platform, since its distributed design and efficient message forwarding.

Among the mainstream messaging middlewares, there are many choices like Apache Kafka, RabbitMQ and

NSQ. RabbitMQ is a relatively an out-dated messaging middleware comparing to NSQ and Kafka, and it has a flaw is dealing with clusters of node. Kafka is widely used by a lot of big firms and use the language Java. We choose NSQ because it is written and implement in Golang, which is a language better for large scale application and provide better scalability comparing to Java. Also, NSQ provides a topology manager nsqlookupd, which can monitor all NSQD brokers and redirect the query of users to NSQD. It has a better scalability comparing to the other two messaging middleware. In this case, we apply nsq service to the second layer of our system.

# 3 Implementation of RSMS

From the Fig.1 system architecture, the implementation of our platform consists of three major parts: data resources, stream processing, message forwarding. The first problem arises when we want to abtain real stock data and convert them to stream. The data resources is from Quandl website, they have different data format either CSV file or json file. Since there doesn't exsits any steam API provided reliable stock data stream, we have to apply kinesis or other stream middlware to provide reliable data stream, which is actually a waste of computation resources because Apache Flink has already provided efficient API to deal with CSV file. Thus, stock data stored as a CSV file, then enter in our system directly processed by Flink CSV reader to create a data stream before those stock data enter in Flink process part.

Stream processing part consists of a Flink cluster and a Flink job with multiple operations. Stock data stream processed by operations will be directly sink to NSQ publishers.

Messaging middleware part include the implementation of NSQD and nsqlookupd. Users query the nsqlookupd for the specific topic and nsqlookupd search over all the NSQD and collect all the messages under the topic and deliver them to the subscriber of that topic.

## 3.1 Apache Flink

Apache Flink is an open-source stream processing framework for distributed, high-performing, always-available, and accurate data streaming applications. Flink is a centralized system with one Job manager responsible for monitoring and controling the state of each task manger where data is being processed. The data transmission inside a Flink job begins with data sources from Job manager to task managers. After processing, flink converges all the data and transmits them back to the Job manger to finish sink operation. It's crucial to deal with the deployment of the job manager and other task managers.

In RSMS, there are three steps to implement a Apache Flink job on cloud. The first step is to choose an appropriate VM to build a cluster based on Amazon EMR cluster. The second step focused on data source and sink, which is actually connectors between stock data resource, flink job manager and data destination. The third step is the most difficult part in Flink, its objective is to apply multiple operation based on RSMS needs. Results show that our implementation of Apache Flink meets real time requirements and data accuracy requirements.

### 3.1.1 Deployment of Amazon EMR

In order to increase processing platform's system load and computation ability, and seperate job manger with multiple task managers, we decided to deploy a cluster with multiple EC2 instances. Since Flink is a centralized system controlled by the job manager, there are map reduce like function required by a cluster running Flink. We eventually decided to use Amazon EMR, a clusters provided by Amazon Web Serives.

Amazon EMR provides a managed Hadoop framework that makes it easy, fast, and cost-effective to process vast amounts of data across dynamically scalable Amazon EC2 instances. Mostly importantly, Amazon EMR successfully applied hadoop yarn framework in order to run Flink upon its cluster. From the Amazon EMR console, we choose c4.xlarge ec2 instances to be the job manager of our cluster, because c4.xlarge is an ideal choice for computational activities. Based on our needs, we deployed two task managers m4.xlarge ec2 types that is for general purpose.

Problem rises when we are trying to read local CSV files which is our stock data resource. We are unable to directly read files stored locally at job manager node. Because Flink programs run in a variety of contexts, standalone, or embedded in other programs, which means its runtime environment is not locally at job manager. Instead its runtime environment is the cluster and we can only read files that is stored at the file system of cluster.

Inspired by the runtime issue, we store our CSV files into HDFS, which is hadoop fils system, similiar to S3 and other distributed file system.

### 3.1.2 Connectors Setup

Connector is an very important concept in Flink. It establish a bridge between data resources and itself, and another bridge between itself and data destination. A few basic data sources and sinks are built into Flink and are always available. The predefined data sources include reading from files, directories, and sockets, and ingesting data from collections and iterators. The predefined data sinks support writing to files, to stdout and stderr, and to sockets.

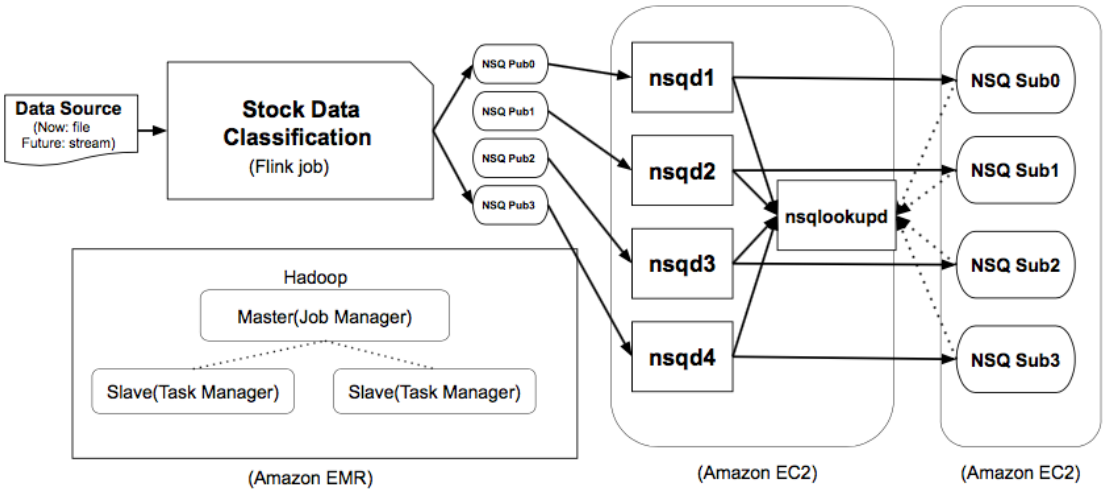


Figure 1: System Architecture

Considered that we need to read stock data from Quandl and transform them into data stream before enter into the cluster. It's an appropriate method to just read the stock data from a CSV file. We apply Dataset¡¿ API to read the stock data. DataSet programs in Flink are regular programs that implement transformations on data sets. Our stock data sets are created from Quandl CSV files. Results are returned via sinks. The execution happens on clusters of many machines.Data sink in our application is directly sink to the NSQ publisher. After tests and analysis, we found that using commandline to connect Flink with NSQ is not very efficient and bad stability. Therefore, we thought about building a connector between Flink and NSQ. Shared memory maybe helpful to build a connector.

#### 3.1.3 Design of Flink Job

Flink job usually consists of mutiple data process operations, the parallelism determines how operations are split into individual tasks which are assigned to task slots. Each node in a cluster has at least one task slot. The total number of task slots is the number of all task slots on all nodes. From programmers' perspective, data process operations are implemented by map functions, inside each map function exits one or more stream process functions. The design of RSMS flink job, based on Stock investors demands of stock data. RSMS's flink job divides into multiple operations, such as Rise stock data filtering, Drop stock data filtering and Date stock data filtering.

### 3.2 NSQ Implementation

Comparing to the design and implementation of first layer of stock data processing and filtering, the NSQ implementation is a basic set up following the official guide of deploying the nsq system. The producer side is deployed on the EMR server. All the information about the stocks under the specific topic is parsed into JSON form and then sent to the NSQD. Then the subscriber on another EC2 can query the nsqlookupd. The nsqlookupd plays an role like a post office with all the NSQDs register their IP address and port in the nsqlookupd side. The NSQD brokers send heartbeats periodically to the nsqlookupd and nsqlookupd once receive query for a specific topic, it redirects the connection to the NSQDs that carry messages about the topic.

The NSQD can be distributed across different servers as long as they all register at the same nsqlookupd. The client side only need to make configuration once for each publisher with the single nsqlookupd, then the nsqlookupd finish the task of redirecting the connection to NSQDs. After the stock information are delivered to the client side, the json data will be reformatted into string for better user experience.

Based on the consideration of the dataset size of our stock data pool, we choose to allocate nsqlookupd and three NSQDs on the same ec2 instance, as shown in Fig.1. Also, we distribute them on different dockers and monitor the activity and CPU utilization of dockers. Each NSQD docker is given different IP ports for TCP connection and the same broadcast address, which is set to the IP address and port number of nsqlookupd. Our implementation environment for NSQ is relatively tiny but can still finish all

the tasks we assign to it. In the real application, the NSQ server cluster can be even huge and can be distributed over multiple physical locations to deal with multiple data stream from different stock exchange over the world.

# 4 Experimental Results

The significance of our Realtime Stockdata Messaging System is that we successfully connect Flink with NSQ to create a real time platform both for data processing and data delivering. In our experiments, we mainly focused on the real time performance of our RSMS. We measured the time interval of each record flows through RSMS as the latency of our system.

The measurement of system latency divides into two parts: Flink latency and NSQ latency. The connector latency is negligible because we jsut simply use command line to pass stock data to the NSQ publisher. The major latency happens on flink cluster's preparation, flink data processing, nsqlookupd registration and nsq data delivering. Therefore, the following experiments will mainly focus on the measurement of flink latency and NSQ latency and our explanation of problems.

## 4.1 Flink Latency Measurements

For the measurement of Flink latency, we want to identify the portion of all different kinds of latency, such as cluster initialization, tasks preparation and data processing. We design several experiments to measure Flink's latency when runs on Amazon EMR cluster. Each experiments contains different size of stock records ranging from 1300 records to 5200 records. Experiment results shows in Fig.2

### 4.1.1 Flink latency Analysis

Experimental results implies that Flink latency can be divided into three parts: event time, process time, ingestion time. Event time represents the time at which event occurred on its producing device. In our system, event time includes not only the data collecting time, but also cluster initialization time. Our experiments shows that the event time spend on data collection and cluster initialization is a constant value, which is nearly 5800 ms. It's not a bad latency since all the platform need some time to prepare for its running.

As Fig.2 shows that flink process time consists of tasks preparation time and operations process time. Tasks preparation happens when flink job manager manage to split each operation into multiple tasks and assignment them to specific task managers. Its value is also a constant number which is close to 200 ms. Operation process time includes multiple operations' operating time. In Fig3, we display process time by first map function processing time and following processing time, each of them represents processing time per 200 records. The reason why we display this model is that the first map operation's process time is significantly longer than the following map operation. That means Flink can easily handle complex flink job and perform real time processing as well. Depends on this analytical results, we found that Each record's processing time varies from 0.5 ms to 4 ms. The total latency of Flink job is the sum of all the three parts. For each record, its average latency is nearly 200ms preparation time plus less than 4 ms process time plus 5800 ms cluster initialization time. Therefore, the most significance of flink is its capability of processing large amount of data with complex operations.

## 4.2 NSQ Latency Measurements

To have a better measurement of NSQ message delivery latency, first, we need to look for a precise time synchronizing tool and then record the time stamp at the proper location in the delivery process. We design and conduct 3 experiments with NSQ latency with different message amount per publisher, different publisher number, and different configuration between subscriber and NSQD.

### 4.2.1 Clock Synchronizing across Servers

In the deployment of Amazon ec2, the NTP clock source is a built-in setup and it is checked by the NTP synchronizer across different time zone periodically. However, NTP across different ec2 might have minor fluctuations, which is crucial in the nanosecond level latency measurement. Thus, we need a more precise time source instead of NTP.

we choose PTPD to synchronize the clock across three servers: the producer server, the broker server and the client server. The accuracy of PTPD is within 1 to 100 microseconds, which is small enough for the purpose of this experiment. Since all the servers are located on Amazon AWS ec2 or EMR (a cluster version of several ec2), we can use Ethernet interface in the configuration of PTPD. In the implementation of PTPD,there are two components, the master clock and the slave clock. The slave clock will listen to the master clock and alter the clock according to the signal sent from the master clock. The master clock is set on the consumer ec2 instance and the slave clock is set on the producer ec2 instance.

In the latency measurement experiment, we encode the current producer system clock time $T_0$ within each message. Then, as soon as the consumer side receive the message, we acquire a current time stamp $T_1$on the consumer server and calculate the latency of message latency by decoding the clock information within the message and

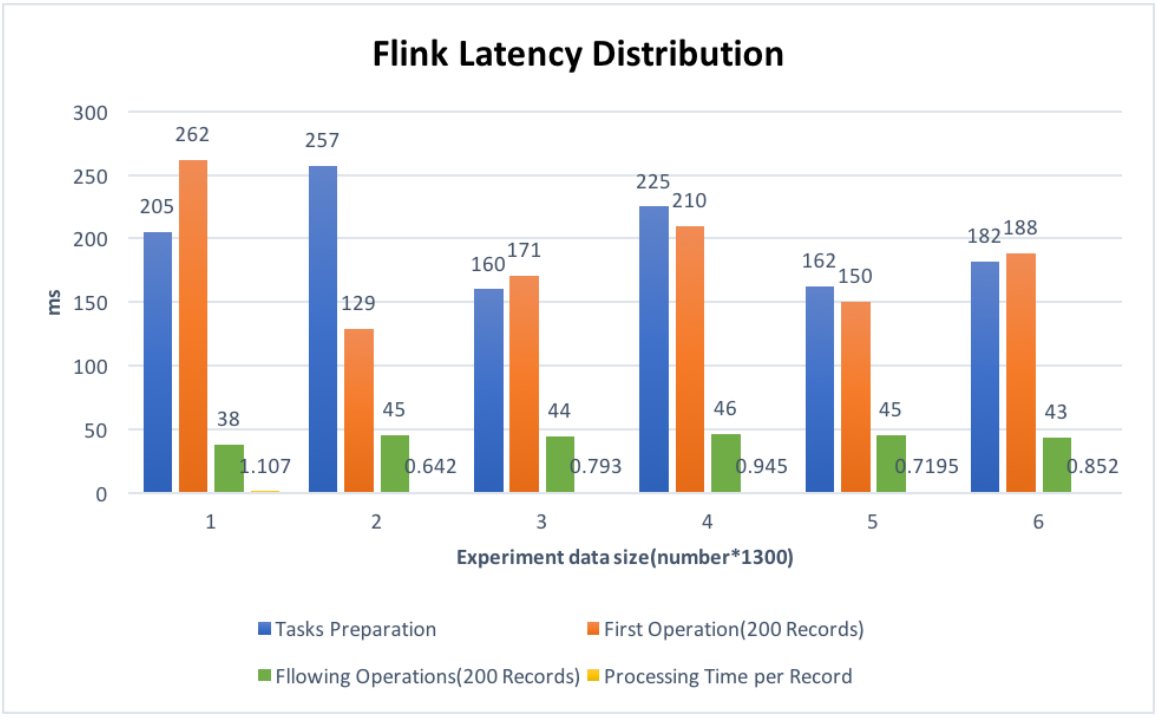


Figure 2: Flink Latency Distribution

$T_1 - T_0$ is the latency of message delivery in the broker middleware. We mark and record the latency valid only if all the messages published in each experiment has been received by the corresponding subscriber. All of latency observed and recorded are valid in the criteria above, which can prove the reliability of our system as well.

#### 4.2.2 Latency and Publisher Number

In this experiment, each publisher produce 3000 messages to 1 topic channel. The publishers are connected with the NSQD server directly and each publisher is assigned We experiment separately with 1 publisher to 20 publishers per producer. There a linear increase of latency as expected. The latency is per message end-to-end latency measured in millisecond level and we can observe the mean of latency increases from 100ms to near 4000ms. When there are more messages in the queue, the broker takes longer to process and deliver the messages.

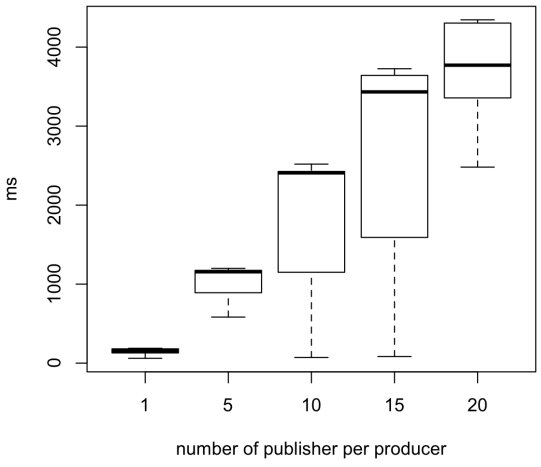


Figure 3: Latency and different publisher number for each producer

#### 4.2.3 Different size of Messages per Publisher

We also measure the delivering latency of messages in different size without nsqlookupd. From the fig.4, we can see that the time consumed increases as the number of messages per publisher multiplied. Rather than multiple the time consumed, it increases less than we expected.

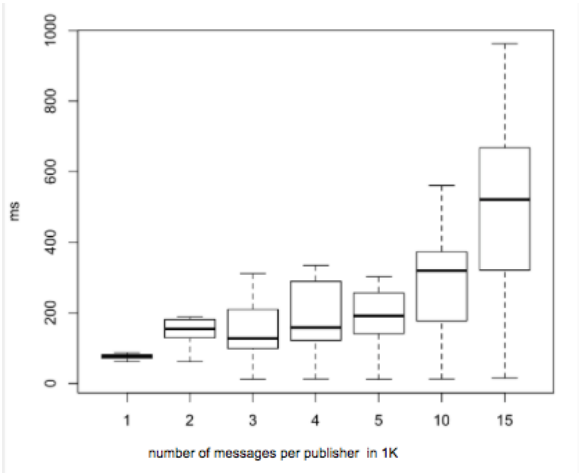


Figure 4: latency for messages in different size

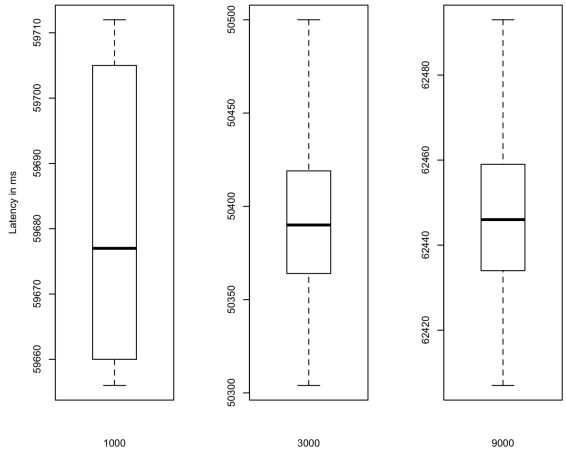


Figure 5: Latency and nsqlookupd with different message amount for each publisher

Thus, we can see that nsq has a good performance on big data processing.

#### 4.2.4 Latency and NSQD/nsqlookupd

NSQD system has a relatively large latency when the producers are connected with nsqlookupd. As Fig.5 shows, the latency for eachwe set the message amount from 1000 to 9000 messages per publisher. We did several experiment with the same configuration but all the experiment reaches a latency of near 60 seconds.

We further investigate into each subscriber's configuration with nsqlookupd and they only query once to the nsqlookupd. After the nsqlookupd found the topic among the NSQDs, no more nsqlookupd connection is made since nsqlookupd redirects all the connections to the specific NSQD. From the comparison with Fig.3, it is not hard to find out that the latency with nsqlookupd connection is too huge and may impede the trading decision.

There is a tradeoff between the scalability nsqlookupd provides and the latency it consumes. For the easy deployment setup of the subscriber, each subscriber only needs to configure with nsqlookupd. However, the connection consume too much time. For further investigation into the nsq source code, we will try to find if the delay of delivery is because of the hardware limitation of nsqlookupd or the bandwidth of ec2 instance in general. For this experiment, we consider nsqlookupd is the bottleneck of our system.

### 4.3 NSQ Scalability and Reliability Measurements

For the scalability purpose, we try to publish 250,000 messages per publisher and the producer has 3 publishers. We monitored the CPU utilization reaches 100%. Every time when one message is delivered, we increase the count of message by one. In the end, the message count reaches 250,000 and we can say that all the messages are marked delivered by the subscriber side. This is the benchmark for NSQ when deployed on the ec2 t2.micro instance when 3 NSQDs and 1 nsqlookupd located on the same server. Also, we prove that NSQ provides a reliable messaging delivery service even when its broker is saturated.

## 5 Conclusion

We proposed a novel real-time stock data processing and messaging system. The platform efficiently combine two layers: the first layer dynamically corporate EMR platform with flink job. We applied Amazon EMR with hadoop Yarn session to run Flink application and connected Flink with NSQ to realize data connection between two individual platform.

The second layer bridges the NSQ producer with EMR platform and have NSQD daemon brokers deployed on a different ec2 server. We realized the topology manager function by using nsqlookupd as the broadcast channel so that the consumers only need to query nsqlookupd instead of remember all the NSQD IP address and ports. The distribution of our system achieves the objective of independence, scalability and reliability. However, the latency of processing and delivery of messages did not achieve our original objectives.

Future work will be devoted to a deeper investigation of the huge latency gap of NSQD. Additionally, we can still maintain the basic structure of our system and improve either the server setup or the bandwidth. To achieve a better experiment result, we would like to deploy a local phys-

ical server and compare its performance with the remote server.