Redesigning Kafka Message Queue System: Toward a Decentralized Stateful Broker System

COEN241 – Cloud Computing
Professor Ming-Hwa Wang
June, 2017

Yiqiao Li, Yue Liu, Sen Zhang, Pengyu Zhu
Department of Computer Science and Engineering
Santa Clara University, CA

Introduction

Modern distributed systems are growing exponentially as far as performance and scale. The sheer complexity and enormity of modern network made it extremely costly to manage node-to-node communication with home-grown systems. Specialized messaging systems, or message queue services, came into being to meet the ever increasing demand on the reliability and performance of message delivery.

Message queue systems today have been and is still evolving from their initial versions, offering mostly services of asynchronous, parallel and distributed capabilities. Most message queue services are distributed themselves in order to keep up with the skyrocketing computing power of their clients. As a system becomes distributed, the issues of inter-process communication, fault tolerance, node organizations and data storing become the focal point of those trying to design a better message queue.

Kafka, initially developed by LinkedIn in 2011, was designed with such performance that shadowed most contemporary peers. It sacrificed some old message queue features such as message ordering, to ensure high-speed message delivery. One of the most important task of node coordination was delegated to Apache's then highly available coordination system, ZooKeeper. ZooKeeper was effective at its job, however, lacks the scalability as most Kafka systems today tend to grow much bigger than its earlier clients.

We believe that ZooKeeper cannot remain an integral part of Kafka if the message queue system were to meet it potential. Kafka needs a more scalable and faster distributed coordination system to breakthrough its already-impressive performance. Therefore, we would like to introduce our alternative architecture for Kafka node coordination system: Decentralized Stateful Broker System (DSBS). We expect that DSBS will offer a scalable and reliable solution to replace ZooKeeper while offer Kafka a boost in message delivery speed.

Theoretical bases and literature review

Some of the predecessors of Kafka was well within the radar of computer scientists. Earlier message queue systems such as RabbitMQ, OpenMQ and ActiveMQ have been subjects of comparison of researches. In 2015 a research named "An Experimental Comparison of ActiveMQ and OpenMQ Brokers in Asynchronous Cloud Environment", by Klein and Stefanescu, conducted an experiment between ActiveMQ and OpenMQ in busy cloud environment with high volume of traffics to compare their performances, message persistence and scalability options. The researchers found that ActiveMQ turns out to be a faster broker in all tested scenarios while also using less memory than OpenMQ.

A different group of researchers, in the same year, conducted experiments to compare ActiveMQ and RabbitMQ, another popular message queue system at the time. Their results showed that ActiveMQ is faster on message reception (the client sends the message to the broker), while RabbitMQ is faster on producing messages (the client receiving messages from the broker).

Kafka was theoretically conceived in an open source project by LinkedIn in early 2011. The paper first introduced how the new message system can be vastly powerful when it comes to message queue performance. Kreps, Narkhede and Rao created Kafka originally as a tool to handle large scale log processing. They introduced a number of unconventional system design to make sure the new system run fast. Kafka outperformed RabbitMQ and ActiveMQ by many times and is proven to consume less resources.

Another paper published in 2015 reexamined the performance and structure of Kafka and proposed additional improvement despite its impressive capabilities. Researchers including Zhenghe Wang and Wei Dai confirmed that Kafka's superior capacity comparing to traditional message queues, but proposed that 1) applications sharing the Kafka system should be able to select processing priorities to reduce suboptimal resource allocations, 2) Kafka need to move away from its heavy dependency on ZooKeeper for node management to increase reliability and system integration, 3) authentication can be added as a feature.

As well known, Kafka currently relies on ZooKeeper, a distributed node coordination managing system, to organize its client and broker information. ZooKeeper is an open source system developed by Apache. Kafka research team used it out of convenience and its good performance. ZooKeeper was first introduced in a research paper, ZooKeeper: Wait-free coordination for Internet-scale systems, by Hunt, Konar, Junqueira and Reed in 2010. It incorporates elements from group messaging, shared registers, and distributed lock services in a replicated, centralized services. ZooKeeper interfaces has the wait-free aspects of shared registers with an event-driven mechanism similar to cache invalidations of distributed files systems.

In 2013, another group of researchers, Skeirik, Bobba and Meseguer, utilized ZooKeeper in a Security-as-a-Service (SecaaS) system. They developed a group key management system and studied its rewriting logic model of a ZooKeeper based group key management service specified in Maude. They focused on the system's fault tolerance and its performance as it scales to service larger grouping using the PVeStA statistical model checking tool.

Despite Kafka and other traditional counterparts, researchers also aimed to study other possibilities when it comes to message queue architectures. In a paper by, Patel, Khasib, Sadooghi and Raicu, they introduced a new message queue system called Hierarchical Distributed Message Queue (HDMQ). The HDMQ system uses a hierarchical structure to organize storages nodes and a round robin algorithm to store and retrieve incoming messages to preserve message ordering, which has been a missing feature in many parallel high-speed message queues. They compared HDMQ across Amazon Simple Queue Service, Windows Azure and IronMQ and discovered that HDMQ outperforms all of them in many aspects.

When evaluating cloud-based message queueing systems (CMQSs), numerous approaches to measure system performance are available, there is no modeling approach for estimating and analyzing performance of CMQSs. In a paper by Li, Cui and Ma, in 2015, they developed a visibility-based modeling approach (VMA) for simulation model using colored Petri nets. Their results reveal considerable insights into resource scheduling and system configuration for service providers to estimate and gain performance optimization.

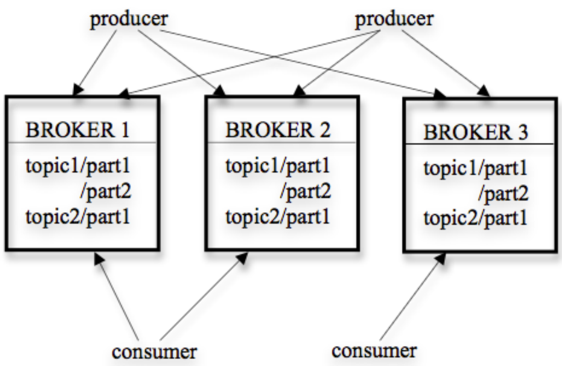
Hypothesis

1. *Decentralized Stateful Broker System with Kafka will result in higher throughput than that using ZooKeeper style system*

Methodology

Our research will focus on a skeleton implementation of the Kafka message queue system. The primary system will be built using Java. The programs will be running and tested on Linux machines. The distributed communication between end nodes within the system will be implemented using sockets with TCP connections.

Kafka message queue system requires three primary entities: Producers, Brokers and Consumers.



Producers: primary data contributors that produce messages and push them into the message queue so data consumers can later retrieve them. Producers directly communicate with one of the brokers in the queuing system and obtain information about message partitioning and split outgoing data and store them to corresponding nodes within the queueing cloud. When storing data, a topic must be established first and the consumers retrieve all data within that topic.

Brokers: primary storage nodes that consists the entire queueing network. They receive data sent from data producers, store them then dispatch them when consumers make requests. In a traditional Kafka broker system, a cluster of machines running ZooKeeper system will maintain the coordination, data partitioning and consumer offset info processing and fault tolerance for all broker nodes.
Consumers: usually request data as consumer groups. Consumers subscribe to a certain topic and retrieve all available messages stored under that topic. Each consumer from a consumer group will receive data from one or more brokers that store messages on the requested topic. The number of consumers cannot be more than the number of partitions granted to that topic.

ZooKeeper Architecture: ZooKeeper acts merely as a node-data information table that dictates 1) which brokers messages under a certain topic are stored, 2) what are the current available brokers, 3) if replica is on, which brokers are leaders and which are backups, 4) at what progress (offsets) have consumers already gone through on each broker.

Decentralized Stateful Broker System: This is our proposed architecture to replace ZooKeeper while increasing Kafka performance. Our design is to keep node coordination information copies in each broker nodes instead of a centralized system such as ZooKeeper. This might increase the time required to update those info as nodes enter and leave the system, but will spread out the workload of a single centralized hub system, thereby reducing the amount of communication necessary to accomplish the tasks.

Here is a comparison between the ZooKeeper paradigm and our stateful broker paradigm:

ZooKeeper Broker Information Table (independent of broker network):

Topic1	Partition1	Broker1	Consumer1.offset Consumer2.offset
	Partition2	Broker2	Consumer1.offset Consumer2.offset
	Partition3	Broker3	Consumer1.offset Consumer2.offset

DSBS Information Tables (on broker1):

Consumer1	Offset
Consumer2	Offset
Consumer3	Offset

Topic1	Partition1	Broker1
	Partition2	Broker2
	Partition3	Broker3

ZooKeeper collectively store all information about each consumer and their partition offsets on each machine, which requires constant update from each broker nodes. When the system simultaneously serves large number of consumer actions on thousands of broker nodes, the influx of information can put heavy burden on the ZooKeeper system in service. On the other hand, our stateful broker model keeps consumer offset information on each individual brokers, without having to communicate with other system, thereby devoting all available bandwidth to data storage from producer and data dispatching to consumers.

Experiment and Testing: we will use one Linux machine as a producer and one additional Linux machine as consumer. Both machines will use multi-thread programing to simulate a producer/consumer group in action instead of using multiple machines to achieve the similar effect. A group of 3-5 broker nodes will be used as the central Kafka storage cluster. The experiment will be divided into two group: test and control group. Test group system will be running our proposed DSBS. All 3-5 broker nodes will be set up to individually have a copy of network information. On the other hand, the control group system will be equipped with a traditional Kafka style structure, with the 3-5 brokers acting only as storage and data senders, while an independent machine act as a ZooKeeper node to manage all node and data administrative information. Once both groups are correctly set up, we will use the producer machine to send the same set of messages, with granularity of size from 1KB to 128KB, to test the sending performance and receiving performance as the messages pass through the test group brokers and the control group brokers then finally reach the consumer machine. The data we will focus on will be throughput and latency. One data is collected, we will conduct statistical analysis and compare the results between two systems.

Implementation

Our implementation consists of four major pieces of Java code: Producer, ZooKeeper Brokers, DSBS Brokers and Consumers. When running data through each set of experiment, we keep the Producers and the Consumers the same and ignorant of the broker system they are dealing with.

Our experiment also has two different scenarios: isolated production/consumption and streaming. When doing isolated P/C, we have the Producer push data to the broker system without a Consumer subscribing at the same time, record production performance, then start the Consumer process, then record its performance. Here is a workflow of our experiment when production/consumption is separate:

When doing streaming experiment, we slightly change the order of events:

Our Producer and Consumer are able to customize the batch size of messages (the number of messages/record transmitted in a single communication package). The Producer can also customize the message size (1KB to 128 KB). The Consumer must specify the number of records/messages consumed as the end of each testing session. All our testing session is set at 30,000 messages, regardless of message size.

Data Analysis and Discussion

Our experiment is divided into two distinct testing condition: isolated production/consumption testing and streaming testing. When conducting the first scenario, we test data production and consumption independently of one and the other, while the streaming scenario have production and consumption process run at the same time, simulating a real life Kafka use case. We also collected data in terms of both the number of records (messages) processed and by Kbps.

Production Throughput Results:

In terms of number of records processed, these are the test results:

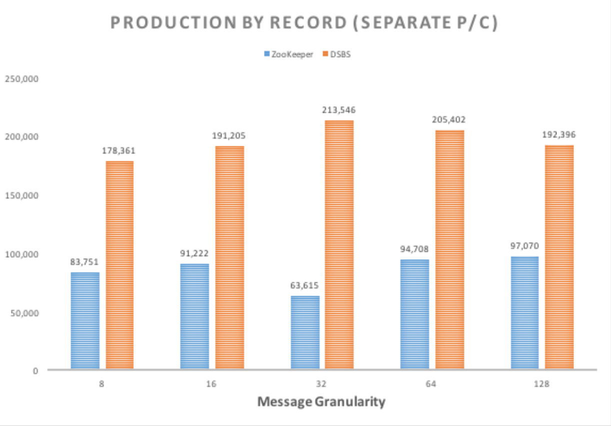


Figure 1

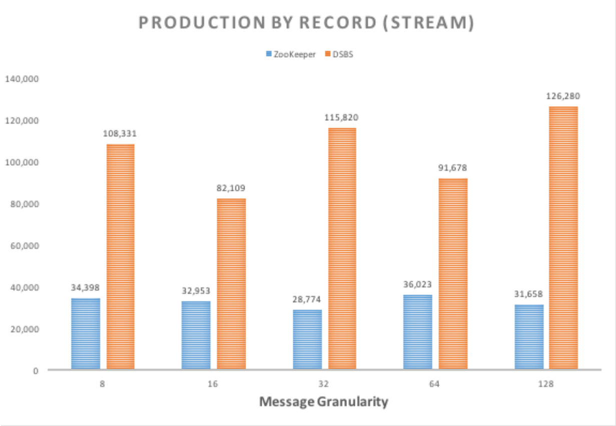


Figure 2

As we can observe, in both separated P/C (Production/Consumption) and Streaming scenarios, in all message granularities, DSBS has higher per record production throughput than Kafka with ZooKeeper. On the other hand, as data granularity increase, the per record production throughput generally remain relatively stable.

In terms of Kbps:

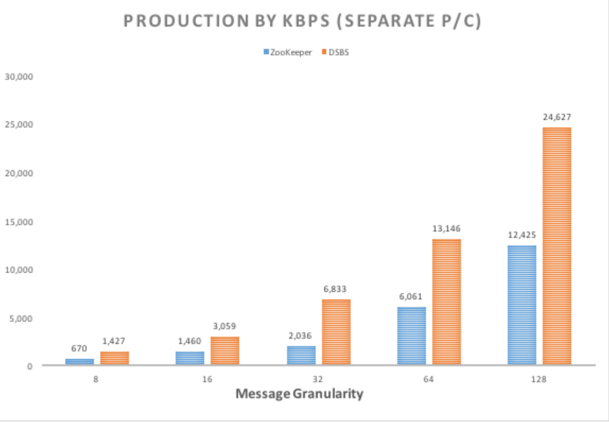


Figure 3

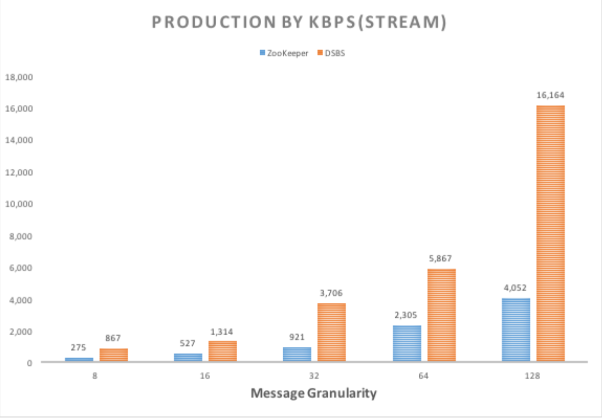
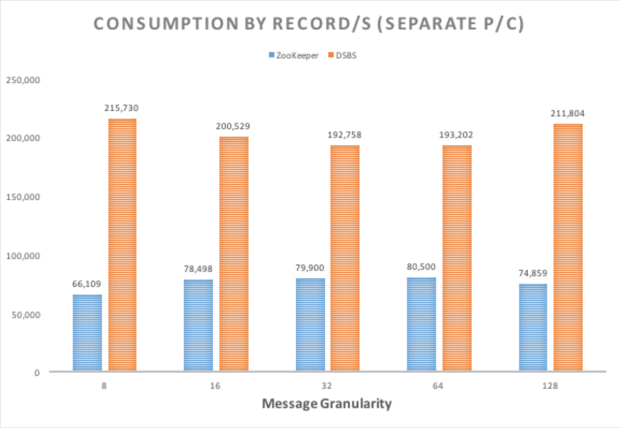


Figure 4

The same trend between DSBS and ZooKeeper remains, while here we see that as message granularity increase, the overall Kbps throughput also increases accordingly.

Consumption Throughput Results:



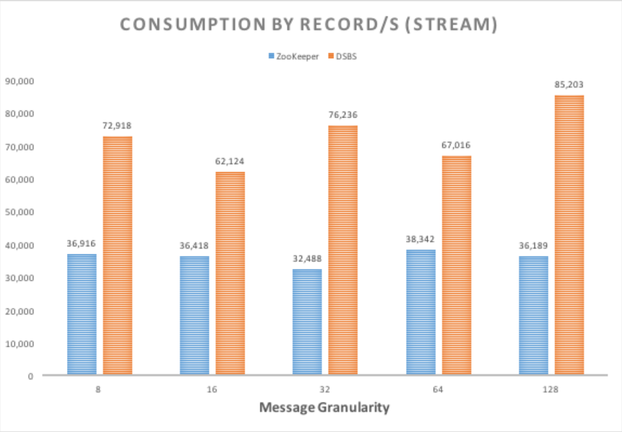
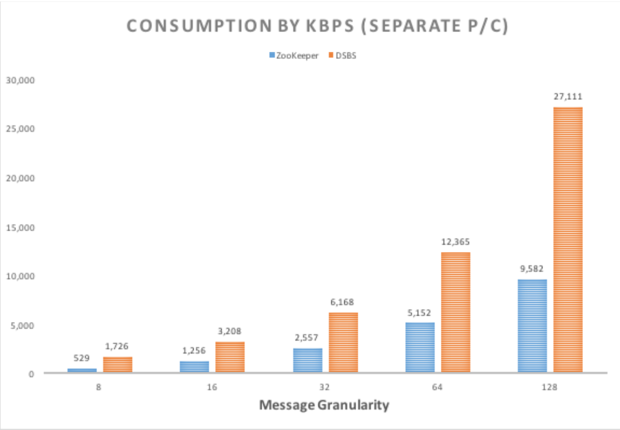


Figure 5 *Figure 6*

Consumption per record results shows generally similar patterns: better performance with DSBS as well as a stable per record throughput across message granularity.



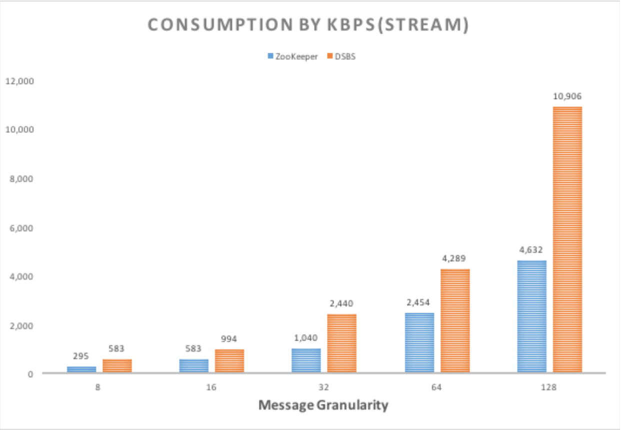


Figure 7 *Figure 8*

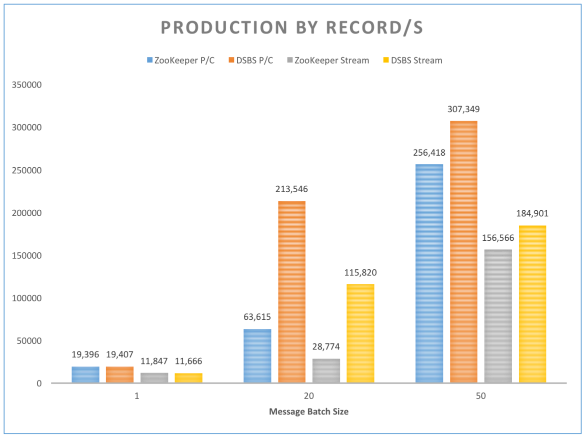
Consumption throughput by Kbps is also similar to production results: higher throughput with DSBS and increasing performance with higher message size.

Here we see the basic trend, on both the production and consumption end, DSBS is out performing ZooKeeper by roughly 2X to 3X as much throughput on both a separated P/C and streaming scenario. Interestingly, we can also observe that message granularity does not seem to affect the per record throughput of either system. No matter how big the message packages are containing, our systems are simply delivering them indifferently at similar speed.

Performance with Varying Batch Size

In addition to what we have above, we also collected result when we keep the message size constant (at 32 byte) while changing the processing batch size (the number of messages/record transmitted in a single communication package).

Production:



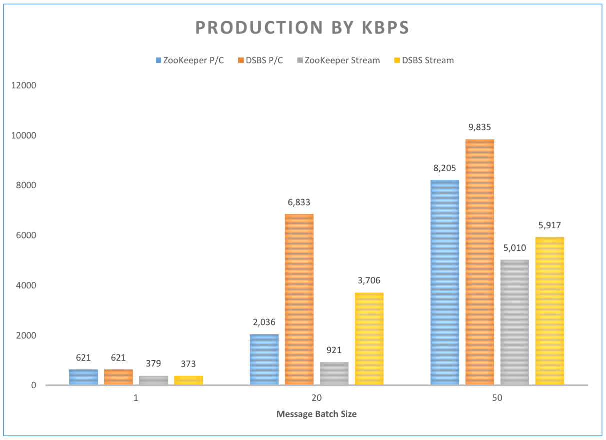
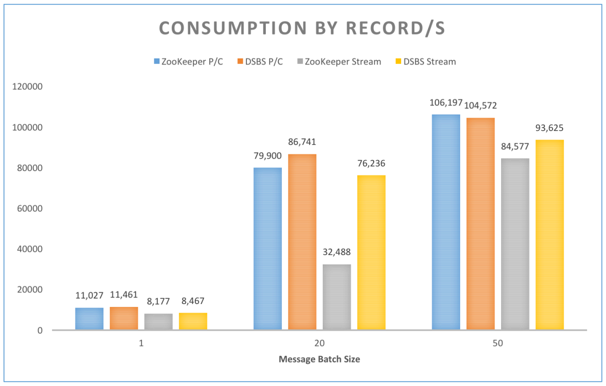


Figure 9

Figure 10

With batch size of 1 record, DSBS and Broker w/ ZooKeeper has similar performance. With increasing batch size, DSBS is delivering higher throughput than Broker w/ZooKeeper on both separate and stream scenarios. In addition, with higher batch size, throughput increases for both systems on separate and stream scenarios.

Consumption:



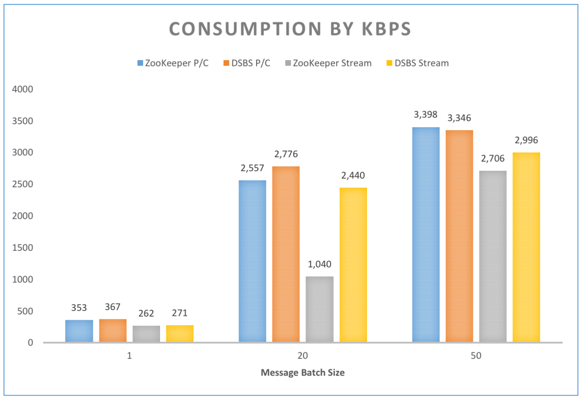


Figure 11

Figure 12

We can observe similar trend here when it comes to consumption performance. But on the separated scenario, the difference of consumption throughput between DSBS and Broker w/ ZooKeeper is not obvious.

Conclusion and Discussion

The Decentralized Stateful Broker System manages to make improvements upon the existing Kafka system with ZooKeeper support. Our hypothesis of DSBS having higher message processing throughput is confirmed across all message granularities that we included in our experiment. By holding both node management and offset information inside each broker instead of storing them in a centralized ZooKeeper, we are able to minimize network traffic necessary to provide fast and large scale distributed message queuing services. At a message batch size of 20, we are able to improve overall throughput by roughly 2X to 3X.

Our experiment illustrates that higher batch size helps to deliver high throughput for both systems. Our observation also confirms the result from the original Kafka paper, which is that a batch delivery can significantly increase the throughput of a message queue. However, the physical hardware limitation may come into play when the batch size reaching some certain number.

In all our test cases, streaming throughput drops 30%~50% from its peak value (test separately for production and consumption). The explanation can be that while handling streaming requests, the possibility of synchronizations between different threads in the message queue significantly increases when producing and consuming happens at the same time. Object lock is placed on the partition which hinders multithread concurrency thus causes a longer latency.

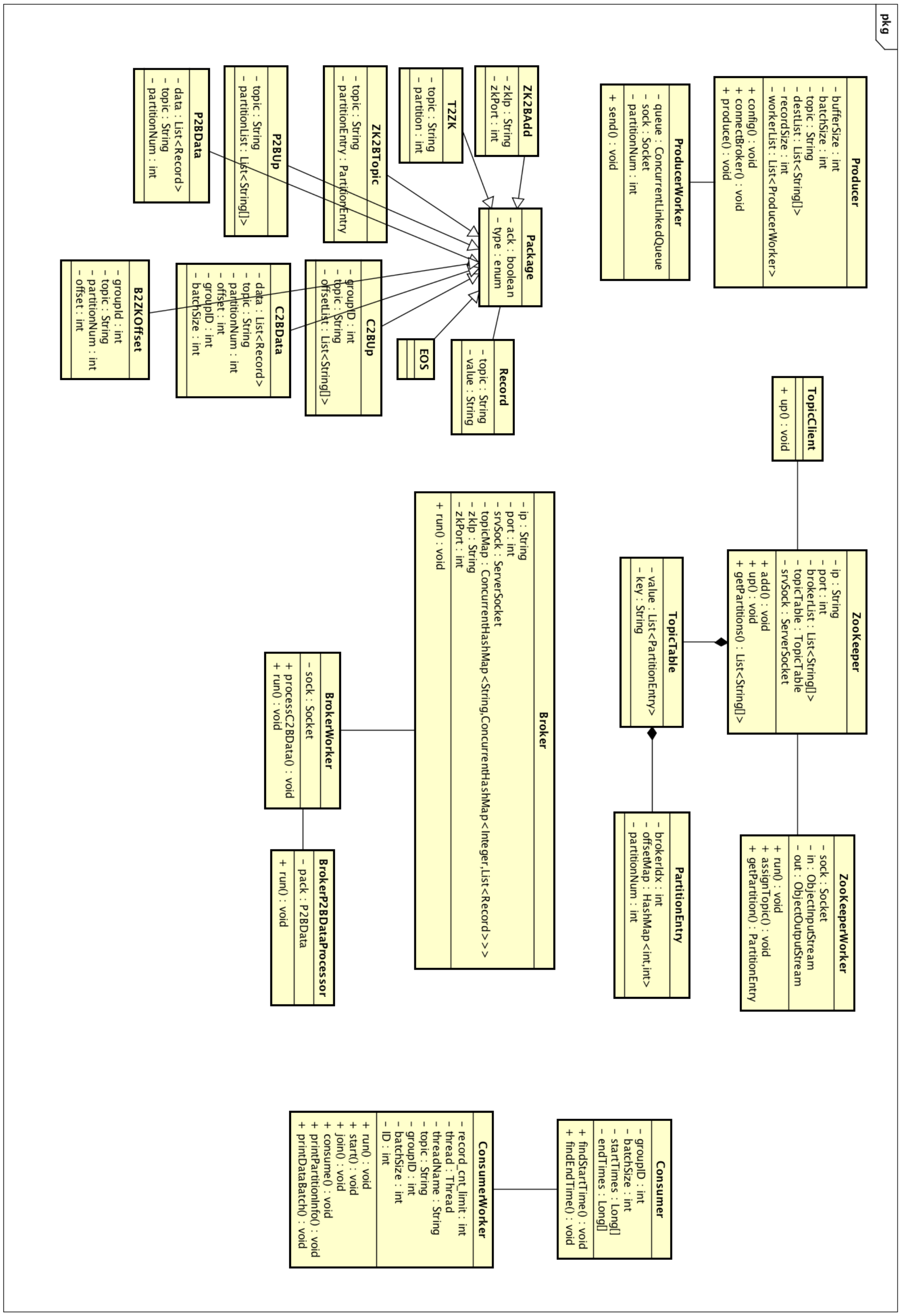
Furthermore, our experiment, due to time and resource constraint, does not fully implement the fault tolerance side of Kafka system. A decentralized node management system will have a rougher time when the system scales up and start to fail from time to time during data transmission. With full degree of replication and possibility of failure, the performance of DSBS might not be as good as what we have in our experiment.

Bibliography

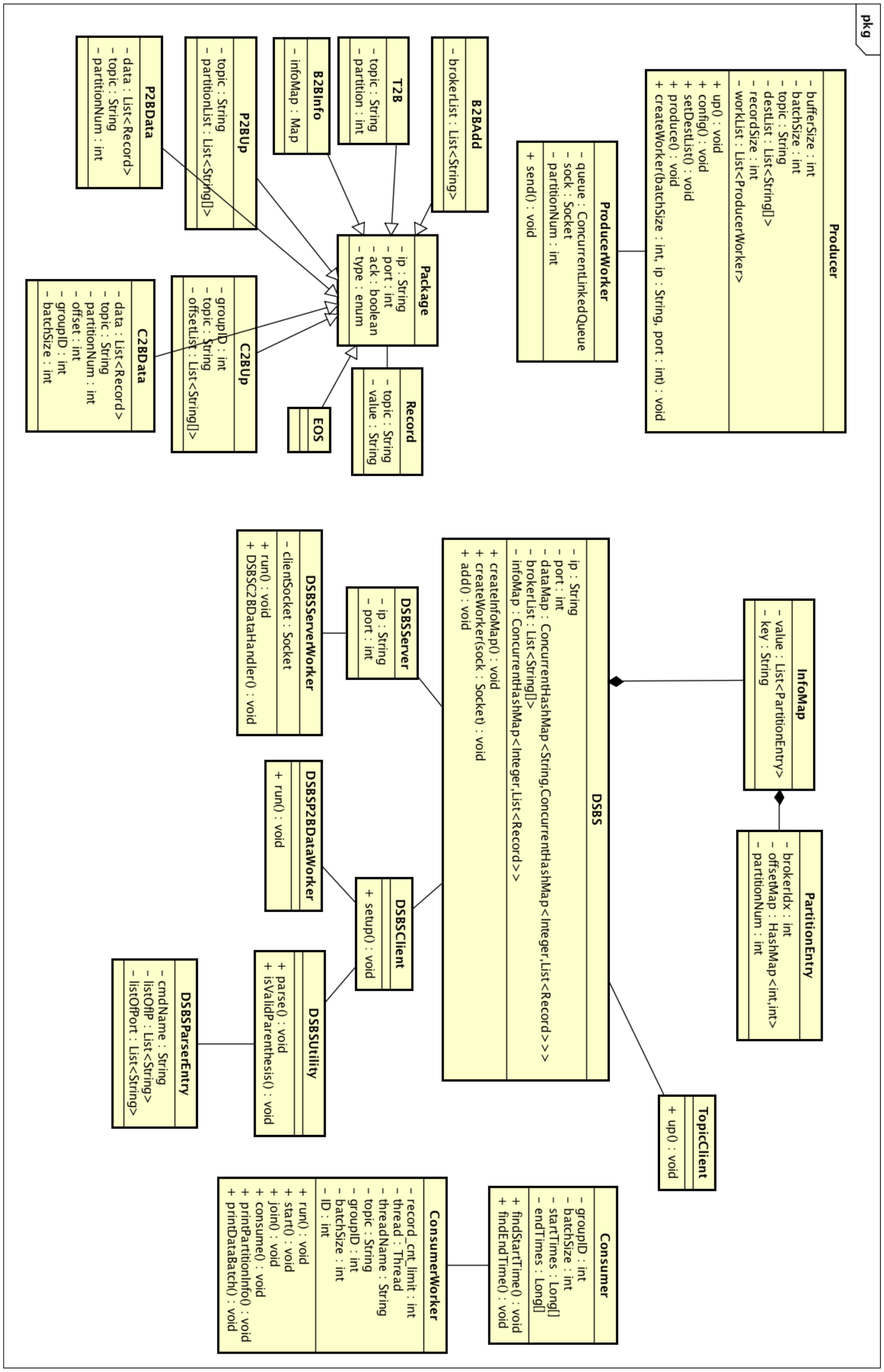
1. Patrick Hunt, Mahadev Konar, Flavio P. Junqueira and Menjamin Reed, ZooKeeper: Wait-free coordination for Internet-scale systems, USENIX Annual Technical Conference, 2010
2. Andrei F. Klein, Mihai Stefanescu, Alan Saied, Kurt Swkhoven, An Experimental Comparison of ActiveMQ and OpenMQ Brokers in Asynchronous Cloud Environment, Digital Information Processing and Communications (ICDIPC), Fifth International Conference, Oct 2015
3. Stephen Skeirik, Rakesh B. Bobba, Jose Meseguer, Formal Analysis of Fault-tolerant Group Key Management Using ZooKeeper, 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2013
4. Dharmit Patel, Faraj Khasib, Iman Sadooghi and Ioan Raicu, Toward In-Order and Exactly-Once Delivery using Hierarchical Distributed Message Queues, Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on, Chicago, IL, 26-29 May 2014
5. Jay Kreps, Neha Narkhede, Jun Rao, Kafka: A Distributed Messaging System for Log Processing, NetDB workshop, 2011
6. Zhenghe Wang, Wei Dai, Feng Wang, Hui Deng, Shoulin Wei, Xiaoli Zhang, Bo Liang, Kafka and its Using in High-throughput and Reliable Message Distribution, Intelligent Networks and Intelligent Systems (ICINIS), 8th International Conference, 2015
7. Jing Li, Yidong Cui and Yan Ma, Modeling Message Queueing Services with Reliability Guarantee in Cloud Computing Environment Using Colored Petri Nets, Mathematical Problems in Engineering, Volume 2015, Hindawi Publishing Corporation, pp. 20.
8. Valeriu Manuel Ionescu, The Analysis of the Performance of RabbitMQ and ActiveMQ, RoEduNet International Conference – Networking in Education and Research, 2015

Appendices

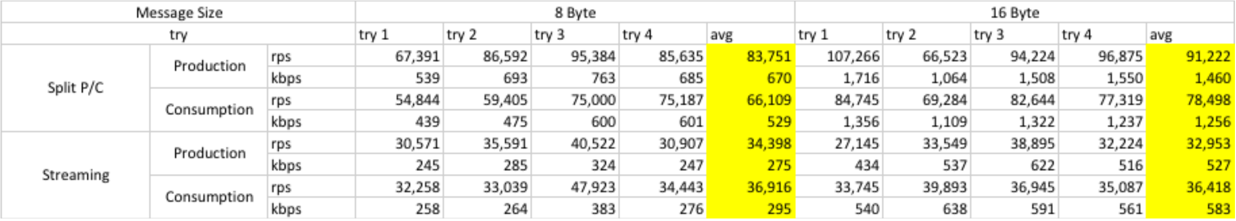
UML: Kafka with ZooKeeper



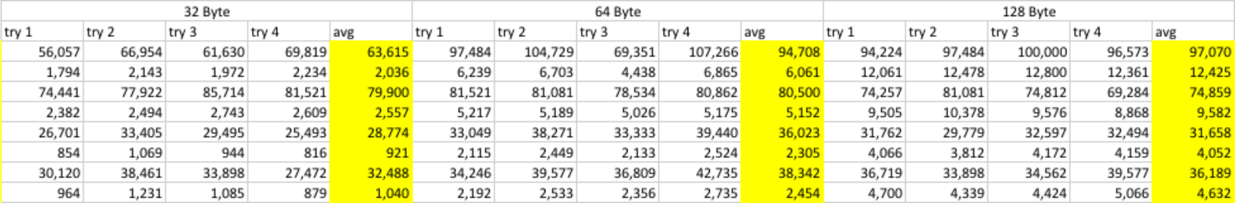
UML: DSBS


pkg
Producer
- bufferSize : int
- batchSize : int
- topic : String
- destList : List<String[]>
- recordSize : int
- workList : List<ProducerWorker>
+ up() : void
+ config() : void
+ setDestList() : void
+ produce() : void
+ createWorker(batchSize : int, ip : String, port : int) : void
ProducerWorker
- queue : ConcurrentLinkedQueue
- sock : Socket
- partitionNum : int
+ send() : void
B2BAdd
- brokerList : List<String>
T2B
- topic : String
- partition : int
B2BInfo
- infoMap : Map
P2BUp
- topic : String
- partitionList : List<String[]>
P2BData
- data : List<Record>
- topic : String
- partitionNum : int
Package
- ip : String
- port : int
- ack : boolean
- type : enum
C2BData
- data : List<Record>
- topic : String
- partitionNum : int
- offset : int
- groupID : int
- batchSize : int
C2BUp
- groupID : int
- topic : String
- offsetList : List<String[]>
EOS
Record
- topic : String
- value : String
InfoMap
- value : List<PartitionEntry>
- key : String
PartitionEntry
- brokerIdx : int
- offsetMap : HashMap<int,int>
- partitionNum : int
DSBS
- ip : String
- port : int
- dataMap : ConcurrentHashMap<String,ConcurrentHashMap<Integer,List<Record>>>
- brokerList : List<String[]>
- infoMap : ConcurrentHashMap<Integer,List<Record>>
+ createInfoMap() : void
+ createWorker(sock : Socket) : void
+ add() : void
DSBSServer
- ip : String
- port : int
DSBSServerWorker
- clientSocket : Socket
+ run() : void
+ DSBSC2BDataHandler() : void
DSBSP2BDataWorker
+ run() : void
DSBSClient
+ setup() : void
DSBSUtility
+ parse() : void
+ isValidParenthesis() : void
DSBSParserEntry
- cmdName : String
- listOfIP : List<String>
- listOfPort : List<String>
TopicClient
+ up() : void
Consumer
- groupID : int
- batchSize : int
- startTimes : Long[]
- endTimes : Long[]
+ findStartTime() : void
+ findEndTime() : void
ConsumerWorker
- record_cnt_limit : int
- thread : Thread
- threadName : String
- topic : String
- groupID : int
- batchSize : int
- ID : int
+ run() : void
+ start() : void
+ join() : void
+ consume() : void
+ printPartitionInfo() : void
+ printDataBatch() : void

ZooKeeper Performance Data:

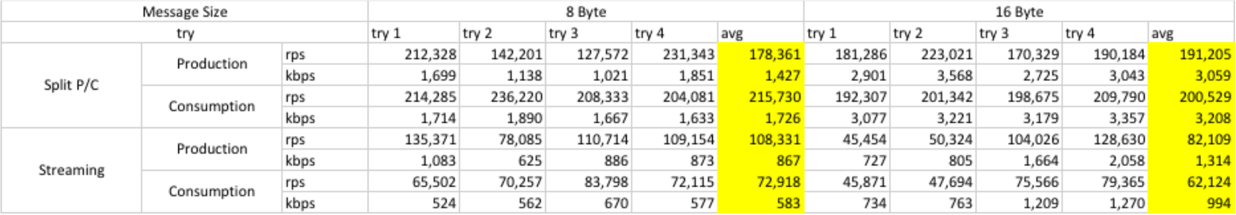


Message Size			8 Byte					16 Byte				
try			try 1	try 2	try 3	try 4	avg	try 1	try 2	try 3	try 4	avg
Split P/C	Production	rps	67,391	86,592	95,384	85,635	83,751	107,266	66,523	94,224	96,875	91,222
		kbps	539	693	763	685	670	1,716	1,064	1,508	1,550	1,460
	Consumption	rps	54,844	59,405	75,000	75,187	66,109	84,745	69,284	82,644	77,319	78,498
		kbps	439	475	600	601	529	1,356	1,109	1,322	1,237	1,256
Streaming	Production	rps	30,571	35,591	40,522	30,907	34,398	27,145	33,549	38,895	32,224	32,953
		kbps	245	285	324	247	275	434	537	622	516	527
	Consumption	rps	32,258	33,039	47,923	34,443	36,916	33,745	39,893	36,945	35,087	36,418
		kbps	258	264	383	276	295	540	638	591	561	583

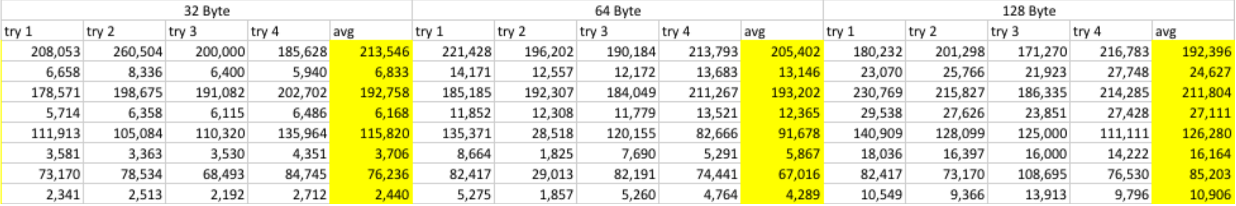


32 Byte					64 Byte					128 Byte				
try 1	try 2	try 3	try 4	avg	try 1	try 2	try 3	try 4	avg	try 1	try 2	try 3	try 4	avg
56,057	66,954	61,630	69,819	63,615	97,484	104,729	69,351	107,266	94,708	94,224	97,484	100,000	96,573	97,070
1,794	2,143	1,972	2,234	2,036	6,239	6,703	4,438	6,865	6,061	12,061	12,478	12,800	12,361	12,425
74,441	77,922	85,714	81,521	79,900	81,521	81,081	78,534	80,862	80,500	74,257	81,081	74,812	69,284	74,859
2,382	2,494	2,743	2,609	2,557	5,217	5,189	5,026	5,175	5,152	9,505	10,378	9,576	8,868	9,582
26,701	33,405	29,495	25,493	28,774	33,049	38,271	33,333	39,440	36,023	31,762	29,779	32,597	32,494	31,658
854	1,069	944	816	921	2,115	2,449	2,133	2,524	2,305	4,066	3,812	4,172	4,159	4,052
30,120	38,461	33,898	27,472	32,488	34,246	39,577	36,809	42,735	38,342	36,719	33,898	34,562	39,577	36,189
964	1,231	1,085	879	1,040	2,192	2,533	2,356	2,735	2,454	4,700	4,339	4,424	5,066	4,632

DSBS Performance Data:

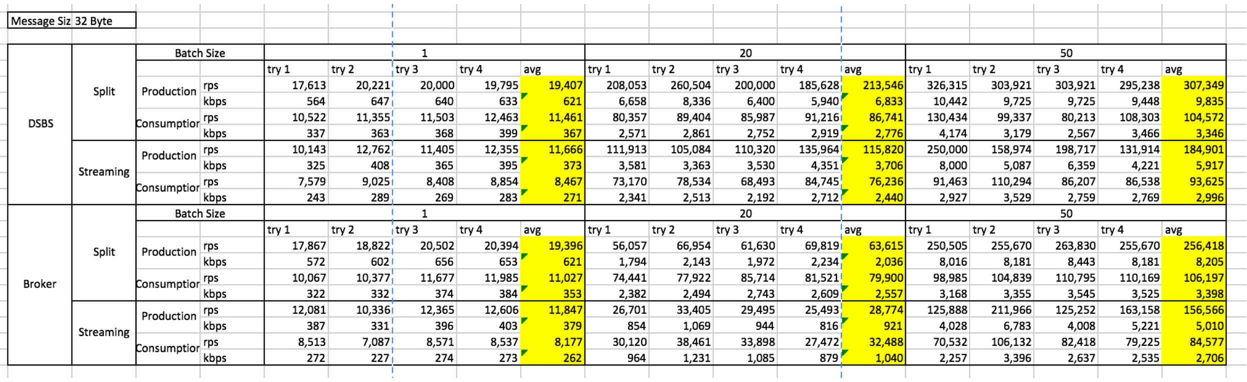


Message Size			8 Byte					16 Byte				
try			try 1	try 2	try 3	try 4	avg	try 1	try 2	try 3	try 4	avg
Split P/C	Production	rps	212,328	142,201	127,572	231,343	178,361	181,286	223,021	170,329	190,184	191,205
		kbps	1,699	1,138	1,021	1,851	1,427	2,901	3,568	2,725	3,043	3,059
	Consumption	rps	214,285	236,220	208,333	204,081	215,730	192,307	201,342	198,675	209,790	200,529
		kbps	1,714	1,890	1,667	1,633	1,726	3,077	3,221	3,179	3,357	3,208
Streaming	Production	rps	135,371	78,085	110,714	109,154	108,331	45,454	50,324	104,026	128,630	82,109
		kbps	1,083	625	886	873	867	727	805	1,664	2,058	1,314
	Consumption	rps	65,502	70,257	83,798	72,115	72,918	45,871	47,694	75,566	79,365	62,124
		kbps	524	562	670	577	583	734	763	1,209	1,270	994



32 Byte					64 Byte					128 Byte				
try 1	try 2	try 3	try 4	avg	try 1	try 2	try 3	try 4	avg	try 1	try 2	try 3	try 4	avg
208,053	260,504	200,000	185,628	213,546	221,428	196,202	190,184	213,793	205,402	180,232	201,298	171,270	216,783	192,396
6,658	8,336	6,400	5,940	6,833	14,171	12,557	12,172	13,683	13,146	23,070	25,766	21,923	27,748	24,627
178,571	198,675	191,082	202,702	192,758	185,185	192,307	184,049	211,267	193,202	230,769	215,827	186,335	214,285	211,804
5,714	6,358	6,115	6,486	6,168	11,852	12,308	11,779	13,521	12,365	29,538	27,626	23,851	27,428	27,111
111,913	105,084	110,320	135,964	115,820	135,371	28,518	120,155	82,666	91,678	140,909	128,099	125,000	111,111	126,280
3,581	3,363	3,530	4,351	3,706	8,664	1,825	7,690	5,291	5,867	18,036	16,397	16,000	14,222	16,164
73,170	78,534	68,493	84,745	76,236	82,417	29,013	82,191	74,441	67,016	82,417	73,170	108,695	76,530	85,203
2,341	2,513	2,192	2,712	2,440	5,275	1,857	5,260	4,764	4,289	10,549	9,366	13,913	9,796	10,906

Varying Batch Size Data:



Message Siz	32 Byte																	
DSBS	Split	Batch Size		1					20					50				
				try 1	try 2	try 3	try 4	avg	try 1	try 2	try 3	try 4	avg	try 1	try 2	try 3	try 4	avg
		Production	rps	17,613	20,221	20,000	19,795	19,407	208,053	260,504	200,000	185,628	213,546	326,315	303,921	303,921	295,238	307,349
			kbps	564	647	640	633	621	6,658	8,336	6,400	5,940	6,833	10,442	9,725	9,725	9,448	9,835
		Consumptior	rps	10,522	11,355	11,503	12,463	11,461	80,357	89,404	85,987	91,216	86,741	130,434	99,337	80,213	108,303	104,572
			kbps	337	363	368	399	367	2,571	2,861	2,752	2,919	2,776	4,174	3,179	2,567	3,466	3,346
	Streaming	Production	rps	10,143	12,762	11,405	12,355	11,666	111,913	105,084	110,320	135,964	115,820	250,000	158,974	198,717	131,914	184,901
			kbps	325	408	365	395	373	3,581	3,363	3,530	4,351	3,706	8,000	5,087	6,359	4,221	5,917
		Consumptior	rps	7,579	9,025	8,408	8,854	8,467	73,170	78,534	68,493	84,745	76,236	91,463	110,294	86,207	86,538	93,625
			kbps	243	289	269	283	271	2,341	2,513	2,192	2,712	2,440	2,927	3,529	2,759	2,769	2,996
Broker	Split	Batch Size		1					20					50				
				try 1	try 2	try 3	try 4	avg	try 1	try 2	try 3	try 4	avg	try 1	try 2	try 3	try 4	avg
		Production	rps	17,867	18,822	20,502	20,394	19,396	56,057	66,954	61,630	69,819	63,615	250,505	255,670	263,830	255,670	256,418
			kbps	572	602	656	653	621	1,794	2,143	1,972	2,234	2,036	8,016	8,181	8,443	8,181	8,205
		Consumptior	rps	10,067	10,377	11,677	11,985	11,027	74,441	77,922	85,714	81,521	79,900	98,985	104,839	110,795	110,169	106,197
			kbps	322	332	374	384	353	2,382	2,494	2,743	2,609	2,557	3,168	3,355	3,545	3,525	3,398
	Streaming	Production	rps	12,081	10,336	12,365	12,606	11,847	26,701	33,405	29,495	25,493	28,774	125,888	211,966	125,252	163,158	156,566
			kbps	387	331	396	403	379	854	1,069	944	816	921	4,028	6,783	4,008	5,221	5,010
		Consumptior	rps	8,513	7,087	8,571	8,537	8,177	30,120	38,461	33,898	27,472	32,488	70,532	106,132	82,418	79,225	84,577
			kbps	272	227	274	273	262	964	1,231	1,085	879	1,040	2,257	3,396	2,637	2,535	2,706