International Journal of Engineering and Advanced Technology (IJEAT)
ISSN: 2249-8958 (Online), Volume-9 Issue-2, December, 2019

Serverless Stream Processing with Elastic Multi-M/M/s/K Queue System

Jagadheeswaran Kathirvel, Elango Parasuraman

Abstract: ***The high throughput - low latency stream processing systems are required to be elastic enough to scale for varying load spike on-demand. However, in the current stream processing systems, the load shedding is observed which impacts the final accuracy. In order to get rid of this issue, the elasticity can be implemented in all kinds of resources involved in the stream processing systems. This paper focuses on providing the elastic scalability in queues and Serverless functions for the event stream processing systems. First, we explain the need of elastic multi-queue with Serverless function in detail for event stream processing, and then will propose an algorithm for elastic scalability of multi-M/M/s/K Queuing with Serverless functions for the efficient stream processing. The experiment result shows that the system scales very well in short span of time with the help of our proposed algorithm. The increased availability in turn helps improving the high processing throughput in low latency.***

Keywords**:** ***Event Stream Processing, Elastic Multi-M/M/s/K Queue, Serverless.***

I. INTRODUCTION

Though the data stream is getting generated everywhere with high velocity and volume, the real-time processing has some latency while producing the output. This latency leads to the backpressure that discards the new events. As valuable insight might exist in any event, those discarded events will have impact in the final outcome. So, in order to give equal importance for getting the accurate result, the stream processing systems are expected to be highly available [1] on-demand. The recent technologies such as virtualization, container, and cloud can help to meet these requirements. This paper explains the need to think about multi-queue systems for event stream processing and focuses on how elastic multi-queue and Serverless [5] computing can be used together for the effective stream processing systems.

Generally, in Queuing theory, single queue with multiple servers is believed to perform better than multiple queues with multiple servers [7] considering cost and other overheads. This is true for the systems where the fixed number of servers are used. However, in cloud computing, since the queues can be elastically provisioned based on the stream load, having multiple queues on-demand will help us meet the higher QOS latency requirements. Serverless computing helps to deploy the business application without worrying about the infrastructure deployments. These systems will auto-scale based on the load; hence the availability also will be high [14]. Also, these systems are designed to function with less processing and memory requirements. Since processing of events also requires less processing capacity and memory [13], the serverless is the best fit for stream processing. The following is the flow of the rest of this paper. Section II will talk about the background, and in section III, we will provide the overview of the solution, and in section IV, the elastic provisioning algorithms will be explained, and the experiment and results will be explained in section V and VI respectively, and the paper will be summarized in Section VII.

II. RELATED WORK

The research on elasticity in stream processing has been getting more attention in the recent past. Bugra et al [1] proposed a system of auto-parallelization that dynamically adjusts the number of parallel channels to achieve the best performance based on changes in the workload. Marangozova-Martin et al [2] proposed multi-level elasticity in stream processing environments with low latency and minimum resources, and Cardellini et al [3] dealt with effective runtime management in terms of placement and replication decisions while considering the application and resource heterogeneity and the migration overhead, so to select the optimal adaptation strategy that can minimize migration costs while satisfying the application QoS requirements. These papers' objective was to achieve the elastic scalability for stream processing systems based on the individual machines or nodes. Similarly, our earlier work [4] proposed adding additional elasticity on Serverless apps in a single queue stream processing system. Mu-Song et al [6] analyzed the state diagram of multi-queue model with finite lengths, and David Raz et al [7] analyzed the fair operation of multi-queue multi-server, and Hedayati et al [8] demonstrates that a single queue system is more fair than multiples queues, and Röger et al [9] proves that multi-queue fair queuing achieves both fairness and high throughput. Gurtov et al [10] demonstrates the need of on-demand servers in real time systems. From the above exploration, it seems that the existing works attempted to solve the high scalability issue with the help of either elastically deploying the individual nodes, or using multi-queue, or using single queue on-demand multi servers approach, but neither attempted to solve the problem with elasticity on multi-queue and Serverless computing. Our proposed algorithms will elastically provision the required numbers of queues and Serverless container instances, whenever there is a huge load and demand for high throughput and low latency requirements.

Revised Manuscript Received on December 30, 2019.
* Correspondence Author
Jagadheeswaran Kathirvel*, Research Scholar, Department of Computer Science, Bharathiar University, Coimbatore, INDIA. Email: jagpro@gmail.com
Elango Parasuraman, Assistant Professor, Department of Information Technology, Perunthalaivar Kamarajar Institute of Engineering and Technology, Karaikal, INDIA. Email: elanalin_74@yahoo.com

III. SOLUTION OVERVIEW

In this section, we will first provide the required background on the queuing theory and Serverless computing that are going to be used in our control algorithms, and then provide the core logic behind our algorithms.

Queuing theory: Kendall's notation [11] helps in defining the model for the queuing system and Little's formula [12] helps in finding out the different steady state properties of the queue. The Kendall's notation for a queue can be generalized as A/B/C/D/E/F. Here, A and B represent the probability distribution of event arrival and event servicing respectively, D represents the capacity of the system, and E represents the maximum number of events in total, and F represents the queuing discipline. A and B can have, either M for a Poisson arrival distribution or exponential interarrival distribution or an exponential service time distribution, D for a deterministic, and G for general distribution. C can have a positive integer value, but mostly non-zero. If it is zero, then it means the queue is open to accept the events, but there are no event processors yet. If it is infinite, then it means it a self-service queue. If D and E are not specified, then they are infinite. F can have FIFO, SIRO, LIFO, or priority, and so on. The following symbols and notations are used for finding out the steady state parameters:

- λ - number of events arriving per unit time
- μ - number of events being serviced per unit time
- s - number of parallel event processors
- ρ - utilization factor of the event processors
- L - number of events in system (waiting and in-service)
- L_q - number of events waiting in queue
- W - waiting time of an event in system (waiting and in-service)
- W_q - waiting time of an event in queue
- P_0 - probability that the system is empty
- P_n - probability that there are n events in system

A queue with single server can be represented as M/M/1. A queue with multiple servers can be represented as M/M/s. A queue with multiple servers with fixed number of queueing capacity can be represented as M/M/s/K. In a system like Serverless computing, the number of event processing container instances provisioned will vary based on system performance parameters from time to time. Hence, it can be mentioned as $M/M/s_l,s_u/K$ for easily differentiating the elastic multi-server queue from the fixed multi-server queue. Here, l and u represent the lower and upper bounds of the containers respectively which are provisioned in a Serverless environment from time to time. Since having infinite queue size is unrealistic, we will focus more on the M/M/s/K queue in rest of this paper. In addition to the general symbols and notations given above, the following are some of the specific symbols and notations for M/M/s/K queue:

- K - capacity of the queue
- P_K - Probability that the system is full
- λ_e - Average rate that the events enter into the system

The formulas for the same are given below that will be used in our control algorithms:

$$P_0 = \left[1 + \sum_{n=1}^{s} \frac{(\lambda/\mu)^n}{n!} + \frac{(\lambda/\mu)^s}{s!} \sum_{n=s+1}^{K} (\lambda/s\mu)^{n-s}\right]^{-1}$$

$$P_n = \frac{(\lambda/\mu)^n}{n!} P_0, \text{for } n = 1,2,\dots s$$

$$= \frac{(\lambda/\mu)^n}{s!\, s^{n-s}} P_0, \text{for } n = s, s+1,\dots K$$

$$= 0, \text{for } n > K$$

$$L = \sum_{n=0}^{s-1} nP_n + L_q + s\left(1 - \sum_{n=0}^{s-1} P_n\right)$$

$$L_q = \frac{P_0 (\lambda/\mu)^s \rho}{s!\,(1-\rho)^2} [1 - \rho^{K-s} - (K-s)\rho^{K-s}(1-\rho)]$$

$$W = L/\lambda_e, \text{where } \lambda_e = \lambda (1 - P_K)$$

$$W_q = L_q/\lambda_e$$

We will use the terms "processor", "event processor", or "container" interchangeably in place of "server" of queuing theory in rest of this paper.

Serverless: The underlying container instances in the serverless function will be provisioned on-demand based on load, invocation method, and other variables [14]. Provisioning more fixed number of event processors will not be efficient, as the utilization factor (ρ) will be low when the arrival rate is less. There is a limit for the maximum number of container instances that can be deployed in a Serverless function. The lower bound can be zero, when there is no event in the queue, though it will have the cold-start issue later when the new events arrive.

It is possible that the concurrent issues may arise when more processors compete each other to fetch the events from the queue. In this case, the multi-queue is the only option [16], though it has some overheads. Each queue in the multi-queue needs to have one or more processors. Having fixed number of multi-queues will lead to cold start and low utilization issues. So, there needs elasticity in provisioning the multi-queues. Whenever a new queue is provisioned, a new Serverless function also needs to be provisioned to process the events in the new queue. This will help to meet the QOS requirement. When the load decreases, the newly provisioned queues and their dependent Serverless function can be removed. The expected number of queues and Serverless functions can be pre-provisioned to avoid the cold start-issues during the actual processing. If more queues and Serverless functions are required, those can be auto-provisioned on-the-fly, when the pre-provisioned ones are fully utilized.

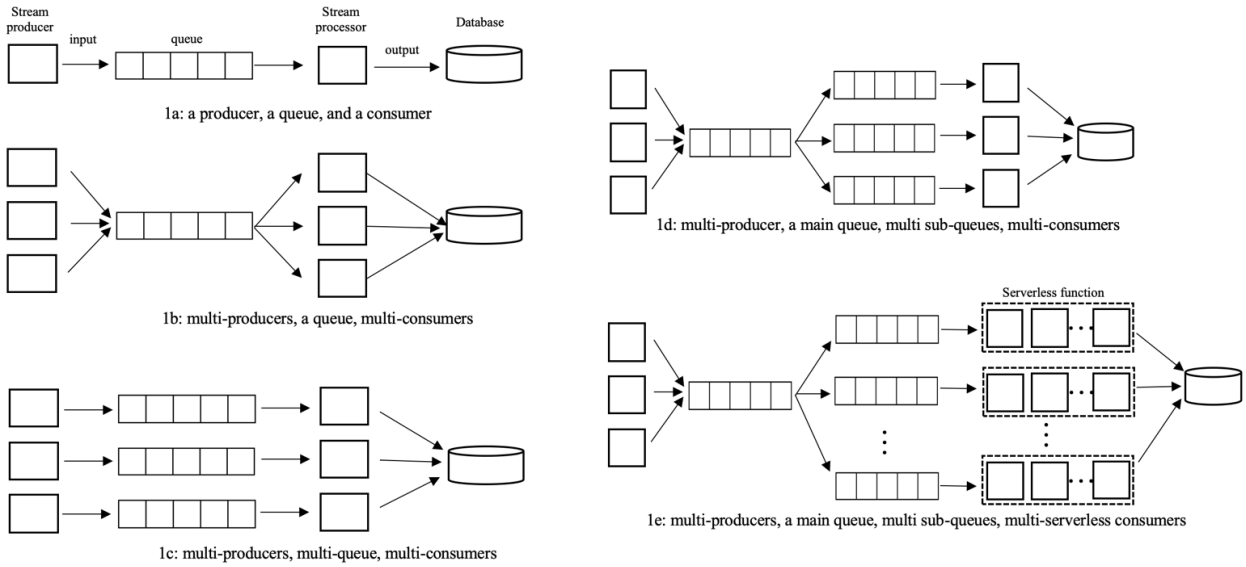

Figure 1: Stream processing systems

The steady state parameter P_K is going to play vital role in this elastic approach. P_K indicates that the probability of the queuing system is full. If it is full, all its dependent variables will have the impact. Figure 1 shows the conceptual design of this paper. A simple stream processing system is in Figure 1a. Figure 1b shows a system where the latency is moderately tolerated. It has multiple producers, multiple consumers on a queue. Figure 1c has multiple producers, multiple queues, multiple processors, and a database. Figure 1d is similar to 1c, but has a main queue in front of the fixed number of sub-queues. Here, the main queue will load balance the events and forward [17] them to sub-queues based on their availability. Figure 1e is similar to 1d, but instead of having the fixed number of queues and fixed number of multiple individually spun up stream processors, these queues will be elastically provisioned. Similarly, the stream processors will be provisioned or de-provisioned on-demand by the Serverless function. Our algorithm will provision the required number of sub-queues and Serverless functions based on the availability of the existing queues.

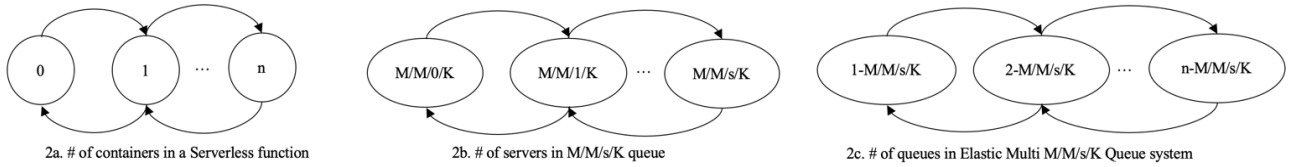

Figure 2: State diagrams of different components of the system

The state diagram of the different components of our proposed elastic system is given in Figure 2. Figure 2a shows the number of containers that gets spun up and spun down in Serverless. When there are no events in the system, the number of containers will become zero. On event arrival, the containers will get provisioned one after the other based on the events arrival rate. It will elastically go up to the maximum number of containers that are allowed in the Serverless function. Figure 2b shows the state diagram of a M/M/s/K queue. It will change based on the number of containers provisioned in Serverless. Figure 2c shows the state diagram of an elastic Multi-M/M/s/K Queue. The new queue is provisioned whenever the existing queues are full, and deprovisioned/released when the load is reduced.

Having elastic multi-queue and elastic processers from Serverless will auto-adjust the event processing system to process with high throughput and low latency requirements. The following are the possible scenarios in this system:

1. One queue – Zero processor: This case will occur when the queue started receiving the events, but there is no processor to process the events yet. Now, the queue is in M/M/0/K state.
2. One queue – One processor: Once the events started adding into the queue, the Serverless function will provision a new processor. There will be cold start latency in this scenario. The queue enters into the M/M/1/K state.
3. One queue – Multi-processors: Based on the arrival rate, the Serverless will keep adding more processors. Then P_K will become low. The queue is in M/M/s/K state now. Average rate (λ_e) of events that enters into the system will be increased. But if there is no reduction in arrival rate, it is time to add the new queues.
4. Multi-Queues – Multi-processors: A new queue and Serverless function is added to reduce the P_K. The system will forward the events to available queues based on P_K of the individual queues. However, P_K of each queue will vary based on the number of operators attached in each queue. The current state of the system is Multi-M/M/s/K. In each cycle, the average P_K of all queues are calculated. The new events are forwarded to the queues based on their P_K.
5. When the arrival rate is reduced, the number of queues will also get reduced, and hence the Serverless function will elastically deprovision the number of event processors that it has spun up.

In a M/M/s/K queue, the main thing to be noted is the queue capacity (K). If it is less than arrival rate and number of event processors, most of the events will be ignored/rejected. This will have impact on the accuracy of the result. The system with finite capacity (M/M/s/K) experiences the frustration and opportunity costs. But in reality, this is the case for all queues, even if the systems are designed to work in M/M/s model because of space limitation in all kinds of resources.

IV. ALGORITHMS

The control algorithm for our elastic Multi-M/M/s/K queue with Serverless processors is given in Algorithm 1. The algorithm starts with required initial infrastructures such as a queue and a Serverless function. There are some thresholds configured. These values will help in scaling the overall system. The scaling algorithm will run at certain frequency, so that the elastic-scaling can happen automatically. Algorithm 2 is for load balancing and forwarding the events to the sub-queues. Algorithm 3 is for pre-provisioning the required number of resources and keeping them in the pool in advance.

The scaling algorithm runs in certain time frequencies and it monitors all queues in this system. It takes care of finding out the required number of queues for processing the stream events in a low latency. There will be one main queue and one sub-queue when the algorithm begins. It finds out the average P_K and average number of Serverless containers created on all queues. If the average P_K is above max P_K threshold and if average number of event processing containers is equal to the maximum possible containers in each Serverless function, then the scale out will happen. The maximum number of containers will vary from one Serverless provider to another, which can be noted either from the provider's documentation or by experience. During scale-out, a new queue will be added from the pre-provisioned resource pool. This pre-provisioned queue would have been attached with a pre-provisioned Serverless function. After the new queue is added, the algorithm will find out the input ratio for each queue to receive the events from the main queue. The events are forwarded based on this ratio by the load balancer and splitter that is given in Algorithm 2.

Similarly, when the average P_K is below the min P_K threshold, the scale-in is initiated, in which a queue is identified for the removal, and its input rate is reduced gradually before finally removing it from the system. The removed queue will go back to the pool of queues which can be reused again when required. But, if the average P_K goes above the min P_K then the scale-in will be cancelled, hence this queue will start receiving more events.

Algorithm 1: Elastic Multi-M/M/s/K Queue Serverless Scaler

```
const maxPk = 0.7
const minPk = 0.4
const minLambda = 100
const maxS = 10

scaler() {
    monitorQueues()

    if (avgPk > maxPk) {
        if(avgS = maxS){
            scaleOut()
        }
    }
    else if (avgPk < minPk) {
        initScaleIn()
    }
    else if (avgPk >= minPk) {
      cancelScaleIn()
    }
}

monitorQueues() {
    avgPk = queues.avg(queue => queue.Pk)
    avgS = queues.avg(queue => queue.S)
    sumS = queues.count() * maxS
}

scaleOut() {
    queues.add(getQueueFromPool())
    generateLambdaRatio()
}

generateLambdaRatio() {
    sumPk = queues.sum(queue => queue.Pk)
    foreach (queue in queues) {
        queue.deltaPk = sumPk - queue.Pk
    }

   sumDeltaPk = queues.sum(queue =>
queue.deltaPk)
   foreach (queue in queues) {
        queue.ratio = queue.deltaPk / sumDeltaPk
   }
}

initScaleIn() {
    if(!hasRemovableQueue()) {
        foreach (queue in queues) {
            if (queue.Pk <= avgPk) {
                queue.removable = true;
                break;
            }
        }
    }

    removableQueue = getRemovableQueue()

    if (removableQueue) {
       reduceLambda(removableQueue);
       if (removableQueue.lambda <= minLambda) {
            scaleIn(removableQueue)
       }
    }
}

scaleIn(queue){
    queues.remove(queue)
    resourcePool.add(queue)
    generateLambdaRatio()
}
```

```
cancelScaleIn() {
    resetRemovableQueue()
    generateLambdaRatio()
}

reduceLambda(queue) {
    queue.lambda = queue.lambda / 2
}

hasRemovableQueue() {
    return getRemovableQueue().count() > 0
}

getRemovableQueue() {
    return queues.select(queue => queue.removable
  == true)
}

resetRemovableQueue() {
    foreach (queue in queues) {
        queue.removable = false;
    }
}

getQueueFromPool() {
    if(resourcePool.count() > 0)
       return resourcePool.firstQueue()
}
```

Algorithm 2: Forwarder

```
loadBalancer() {
    mainQueueLambda = mainQueue.lambda
    removableQueue = getRemovableQueue()
    if (removableQueue){
        mainQueueLambda =
          mainQueue.lambda -
removableQueue.lambda

    foreach (queue in queues) {
       if(queue!=removableQueue)
       queue.lambda = queue.ratio *
mainQueuelambda
    }
}
```

Algorithm 3: Pre-provisioner

```
resourcePoolMonitor() {
  const minQueues = 2
  if (resourcePool.availableQueues() < minQueues)
{
        newQueue = new Queue()
        newServerlessFunc = new ServerlessFunc()
        newQueue.Add(newServerlessFunc)
        resourcePool.addQueue(newQueue)
    }
}
```

V. IMPLEMENTATION

The experiment was performed in Azure cloud. The experimental setup had data producers, queues, and function app. The synthetic data that was produced by the multiple producers were added into the Queue. The Function App was created to classify the transactions with simple logic. The classification result was stored into the Azure Table along with its processing time and transaction details. This output table was used to identify the final accuracy. The main queue was getting filled with the data added by the producers. Initially there was only one sub-queue. The function app attached to the sub-queue started processing the transactions. Since there were huge number of transactions had been being added into main queue, the new event processing instances were added by the first Function App one by one. At certain stage, it was noticed that the number of outgoing items were very less than the incoming items, and eventually the load shedding started occurring. Then our algorithms that were scripted with Azure CLI were executed. The algorithm subsequently picked one of the pre-provisioned queues from the pool, and the load balancer started forwarding the events to both the queues. The containers of both queues' Function App started processing the transactions. Subsequently the algorithm created few more sub queues, since there were huge transactions that were yet to be processed in the main queue. After some time, the number of outgoing messages were increased in all sub queues, and load shedding was gotten rid of completely.

VI. RESULT

The experimental results are shown in this section. The Table 1 and 2 show the steady state parameters of event stream processing system with single queue - multi servers and elastic multi queues - multi-server systems respectively. The λ, μ, and K are in the scale of thousands. In single queue system the λ, μ, and K are 200, 10, and 50 respectively. In elastic multi-queue system, μ and K remained the same as single queue, but λ was reduced to 100.

Table 1. Single queue – multi servers

Time	s	P_0	P_K	$\lambda^*(1-P_K)$	λP_K	L_q	L	W_q	W	$\lambda/(s\mu)$	ρ
1	1	0.00	0.95	10.00	190.00	48.95	49.95	4.89	4.99	20.00	1.00
2	2	0.00	0.90	20.00	180.00	47.89	49.89	2.39	2.49	10.00	1.00
3	3	0.00	0.85	30.00	170.00	46.82	49.82	1.56	1.66	6.67	1.00
4	4	0.00	0.80	40.00	160.00	45.75	49.75	1.14	1.24	5.00	1.00
5	5	0.00	0.75	50.00	150.00	44.67	49.67	0.89	0.99	4.00	1.00
6	6	0.00	0.70	60.00	140.00	43.57	49.57	0.73	0.83	3.33	1.00
7	7	0.00	0.65	70.00	130.00	42.46	49.46	0.61	0.71	2.86	1.00
8	8	0.00	0.60	80.00	120.00	41.33	49.33	0.52	0.62	2.50	1.00
9	9	0.00	0.55	90.00	110.00	40.18	49.18	0.45	0.55	2.22	1.00
10	10	0.00	0.50	100.00	100.00	39.00	49.00	0.39	0.49	2.00	1.00
11	10	0.00	0.50	100.00	100.00	39.00	49.00	0.39	0.49	2.00	1.00
12	10	0.00	0.50	100.00	100.00	39.00	49.00	0.39	0.49	2.00	1.00

Table 2. Multi queues – multi servers

Time	s	P_0	P_K	$\lambda^*(1-P_K)$	λP_K	L_q	L	W_q	W	$\lambda/(s\mu)$	ρ
1	1	0.00	0.90	10.00	90.00	48.89	49.89	4.89	4.99	10.00	1.00
2	2	0.00	0.80	20.00	80.00	47.75	49.75	2.39	2.49	5.00	1.00
3	3	0.00	0.70	30.00	70.00	46.57	49.57	1.55	1.65	3.33	1.00
4	4	0.00	0.60	40.00	60.00	45.33	49.33	1.13	1.23	2.50	1.00
5	5	0.00	0.50	50.00	50.00	44.00	49.00	0.88	0.98	2.00	1.00
6	6	0.00	0.40	60.00	40.00	42.50	48.50	0.71	0.81	1.67	1.00
7	7	0.00	0.30	70.00	30.00	40.67	47.67	0.58	0.68	1.43	1.00
8	8	0.00	0.20	80.00	20.00	38.00	46.00	0.48	0.58	1.25	1.00
9	9	0.00	0.10	89.91	10.09	32.39	41.38	0.36	0.46	1.11	1.00
10	10	0.00	0.02	100.00	0.00	18.64	28.64	0.19	0.29	1.00	1.00
11	10	0.00	0.02	100.00	0.00	18.64	28.64	0.19	0.29	1.00	1.00
12	10	0.00	0.02	100.00	0.00	18.64	28.64	0.19	0.29	1.00	1.00

The charts in Figure 3 shows the comparison between different steady state parameters. The chart in figure 3a shows the arrival rate over time in a single queue system. The average number of events that entered into the system is very less than the average number of events that didn't enter the system. When time flies, the serverless function app started added new containers that lead to the gradual increase in the events' entry, but when the function app's maximum possible containers are reached, the load shedding continued to exist. Figure 3b shows the arrival rate over time in a multi queue system. Here, the overall load got load balanced into sub queues. The average number of events that entered into the system was very less than the average number of events that didn't enter the system.

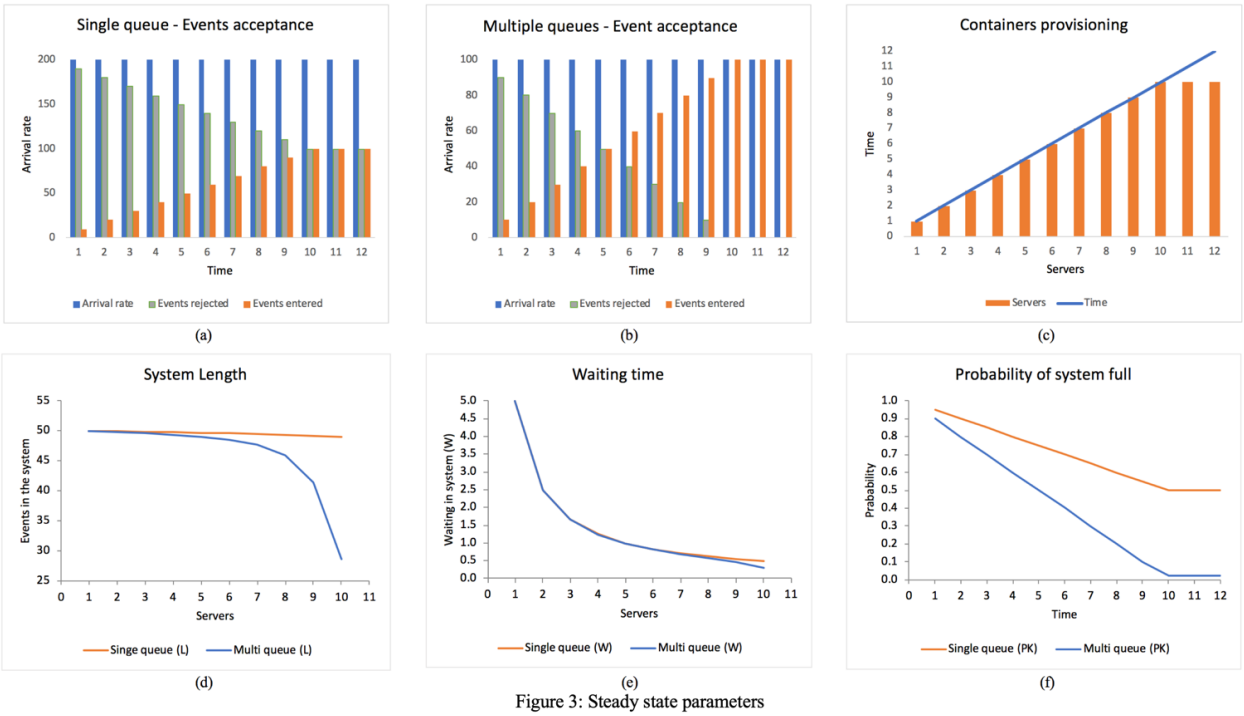

Figure 3: Steady state parameters

After the Function App started adding the new containers, there was a gradual increase in the entry. Since the new queues are provisioned whenever the Function App's maximum possible containers are reached, the over-shedding has been completely avoided. Figure 3c shows the containers that are provisioned over time in a Serverless Function App. Figure 3d shows the queue length over single queue vs multi queues. In single queue, the queue length stays higher all the time, but in multi-queue, it reached to lower value, which leads to lower waiting time in Figure 3e. Figure 3f shows the probability of system being full. The probability of system being full is very low in multi-queue than in single queue, so the load-shedding will be rare, which leads to more throughput and higher accuracy in the final result.

VII. CONCLUSION AND FUTURE WORK

We presented a novel elastic stream processing system that scales well with elastic Multi-M/M/s/K queue and Serverless functions. It is observed from the experimental result that our algorithm gets rid of the load shedding issue by adjusting the number of queues and processors elastically. This leads the system to be capable of handling the events with the high processing throughput with low latency. This makes our elastic multi-queue multi-server algorithm to perform better than singe-queue multi-server system. We are going to explore our algorithm in applications where the elastic scalability is highly required on-demand in short notice.

REFERENCES

1. B. Gedik, S. Schneider, M. Hirzel and K. Wu, "Elastic Scaling for Data Stream Processing," in IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 6, pp. 1447-1463, June 2014. doi: 10.1109/TPDS.2013.295
2. V. Marangozova-Martin, N. de Palma and A. El Rheddane, "Multi-Level Elasticity for Data Stream Processing," in IEEE Transactions on Parallel and Distributed Systems, vol. 30, no. 10, pp. 2326-2337, 1 Oct. 2019. doi: 10.1109/TPDS.2019.2907950
3. Cardellini, V, Lo Presti, F, Nardelli, M, Russo Russo, G. Optimal operator deployment and replication for elastic distributed data stream processing. Concurrency Computat Pract Exper. 2018; 30:e4334. https://doi.org/10.1002/cpe.4334
4. Kathirvel, J., & Parasuraman, E. (2019). A QoS-Latency Aware Event Stream Processing with Elastic-FaaS. Volume-8 Issue-10, August 2019, International Journal of Innovative Technology and Exploring Engineering, 8(10), 3756–3762. doi: 10.35940/ijitee.j9965.0881019
5. Stefan Brenner and Rüdiger Kapitza. 2019. Trust more, serverless. In Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR '19). ACM, New York, NY, USA, 33-43. DOI: https://doi.org/10.1145/3319647.3325825
6. Mu-Song Chen & Hao-Wei Yen (2012) A state diagram analysis of the multi-queue M/M/1 model with finite lengths, Journal of the Chinese Institute of Engineers, 35:2, 165-179, DOI: 10.1080/02533839.2012.638514
7. David Raz, Benjamin Avi-Itzhak, and Hanoch Levy. 2005. Fair operation of multi-server and multi-queue systems. In Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS '05). ACM, New York, NY, USA, 382-383. DOI=http://dx.doi.org/10.1145/1064212.106426
8. Hedayati, Mohammad, Michael L Scott, and Mike Marty. "Multi-Queue Fair Queuing," October 2018. http://hdl.handle.net/1802/34380.
9. Röger, Henriette, and Ruben Mayer. "A Comprehensive Survey on Parallelization and Elasticity in Stream Processing." ACM Computing Surveys 52, no. 2 (2019): 1–37. https://doi.org/10.1145/3303849.
10. Gurtov, A., & Mazalov, V. (2012). Queueing System with On-Demand Number of Servers. Mathematica Applicanda, 40(2). doi:10.14708/ma.v40i2.358
11. Queuing theory tutorial, https://people.revoledu.com/kardi/tutorial/Queuing
12. Queueing theory formulas, http://irh.inf.unideb.hu/user/jsztrik/education/09/english/index.html
13. Batch Processing vs Real Time Processing – Comparison, https://data-flair.training/blogs/batch-processing-vs-real-time-processing/
14. Azure Functions scale and hosting, https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale
15. Azure Service Bus Management, https://github.com/Azure-Samples/service-bus-dotnet-management/blob/master/src/service-bus-dotnet-management
16. Best practices for improving performance using Azure Service Bus, https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-performance-improvements
17. Auto-forwarding Azure Service Bus messaging entities, https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-auto-forwarding

AUTHORS PROFILE

Jagadheeswaran Kathirvel is pursuing his doctorate in Department of Computer Science at Bharathiar University, India. His area of interests includes data stream processing, data mining, artificial intelligence, along with event driven software architecture, design, and engineering. He completed his master's degree in computer applications in 2007 at Bharathiar University, and bachelor's degree in computer science at Periyar University, India, in 2003.

Elango Parasuraman is working as an Assistant Professor in Department of Information Technology at Perunthalaivar Kamarajar Institute of Engineering and Technology, Karaikal, India. His area of interests includes image processing, data mining, and web mining. He completed his Ph.D., at National Institute of Technology Tiruchirappalli, India, in 2011, and his M.Tech., at National Institute of Technology Karnataka, India, in 2005.