An Elastic and Scalable Topic-Based Pub/Sub System Using Deep Reinforcement Learning

Thanos Giannakopoulos, Vana Kalogeraki

▶ To cite this version:

Thanos Giannakopoulos, Vana Kalogeraki. An Elastic and Scalable Topic-Based Pub/Sub System Using Deep Reinforcement Learning. 17th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), Jun 2022, Lucca, Italy. pp.167-183, 10.1007/978-3-031-16092-9_11 . hal-04827161

HAL Id: hal-04827161
https://inria.hal.science/hal-04827161v1

Submitted on 9 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

This document is the original author manuscript of a paper submitted to an IFIP conference proceedings or other IFIP publication by Springer Nature. As such, there may be some differences in the official published version of the paper. Such differences, if any, are usually due to reformatting during preparation for publication or minor corrections made by the author(s) during final proofreading of the publication manuscript.

An Elastic and Scalable Topic-Based Pub/Sub System Using Deep Reinforcement Learning

Thanos Giannakopoulos and Vana Kalogeraki(✉)

Department of Informatics, Athens University of Economics and Business Athens, Athens, Greece
{thanos,vana}@aueb.gr

Abstract. The ability to handle large volumes of event data and react to unexpected spikes, in real-time, remains an important challenge in stream processing systems, such as Apache Kafka, due to the amount of custom coding and technical expertise required to configure these systems. In this paper we investigate the use of reinforcement learning as a promising approach to address these issues. By feeding the machine learning technique with system performance metrics under a wide variety of configurations, we can effectively address any changes in the pub/sub system or overload situations while maintaining the desired performance goals. We implement our methodology on the Kafka pub/sub system without any changes in the application logic. Our experimental results illustrate the performance and benefits of our approach.

Keywords: elasticity · pub/sub · deep reinforcement learning

1 Introduction

Pub/sub systems have been increasingly popular communication architectures in recent years to achieve information dissemination between a set of loosely coupled producers (also known as publishers) and consumers (also known as subscribers). Publishers forward their publications to a set of brokers which are then responsible to deliver the publications to subscribers based on the registered subscriptions. They have found application in a wide variety of domains from online games [1] to stock trading [2]. Examples of popular pub/sub systems include Facebook's Wormhole [3], Google's Cloud Pub/Sub [4], Apache's Kafka [5] and Apache's Pulsar [6].

While pub/sub systems present desirable features including scalability, persistency and availability, several challenges emerge as they typically entail a large number of configuration parameters, which makes their tuning a significant issue that overwhelms users to achieve the desirable performance. Identifying a deployment configuration that satisfies user-defined objectives (e.g.,

© IFIP International Federation for Information Processing 2022

Published by Springer Nature Switzerland AG 2022
D. Eyers and S. Voulgaris (Eds.): DAIS 2022, LNCS 13272, pp. 167–183, 2022.
https://doi.org/10.1007/978-3-031-16092-9_11

on execution time), while avoiding unnecessary over-provisioning, is a significant challenge especially when these are deployed in cloud environments. On the other side, under-provisioning, i.e., allocating fewer resources than required, may lead to services that cannot meet the service level requirements set by the client, and thus must be avoided. Furthermore, pub/sub systems are often deployed in diverse and often dynamic environments, and mechanisms used to ensure robust operation and performance for one environment configuration may not be appropriate for another configuration. While tunable policies provide fine-grained control, configuring a large number of parameters can be overwhelming for users. As a result, users often accept the default settings. Finally, manually tuning requires in-depth knowledge of both the applications and the environment. Nevertheless, it is labor-intensive, time-consuming, and often leads to suboptimal decisions.

The popularity of Machine Learning has grown significantly in recent years as it provides systems with the ability to learn and enhance their operation automatically. In particular, Reinforcement learning (RL) is a type of machine learning algorithm that enables software agents and machines to automatically evaluate the optimal behavior in a particular environment to learn what to do or how to map situations to actions, so as to maximize a numerical reward signal and improve its efficiency. The learner is not told which actions to take, but instead must discover which actions yield the highest reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics, *i.e.,* trial-and-error search and delayed reward, are the two most distinguishing features of reinforcement learning. Reinforcement learning has important benefits over supervised learning as it does not need labeled input/output pairs to be presented and over unsupervised learning which is typically about finding hidden structure and correlations between a set of unlabeled data. State-of-the-art learning approaches can rely on either online or offline schemes to find a (near) optimal configuration. In our work we focus on *online approaches* that can work efficiently even when no a priori knowledge of the execution environment or the deployed application is available.

In this paper we present our approach for building a robust, scalable and elastic pub/sub system utilizing Deep Reinforcement Learning techniques. Our solution addresses the challenges outlined above by combining the following novel features: We investigate Deep Reinforcement Learning as a promising approach to addressing these issues in environments where the dynamics of the environment and the rewards at each state may change and are not necessarily known in advance. By feeding the machine learning technique with system performance metrics under a wide variety of configurations and conditions, we can efficiently address any workload changes or overload situations in the pub/sub system and maintain the desired performance goals. We investigate two different algorithms, namely Deep Q Networks and Double Deep Q Networks where the goal is to learn a policy to tell the Deep Learning Reinforcement agent what action to take under what circumstances. We implement our approach on top of Apache Kafka, a general-purpose pub/sub system, without the need to modify its architecture

or change the producer and consumer application logic. Our approach implements elasticity by dynamically determining whether brokers need to be added or removed from the Kafka cluster based on user demands (*i.e.,* on execution time). We collect a set of performance metrics via the Prometheus monitoring tool, the collected statistics are sent to the Deep Reinforcement Learning Agent to decide the appropriate scaling action. Our experimental evaluation illustrate the performance, scalability and elasticity of our approach.

2 System Architecture and Model

In this section we first present a brief introduction to the Kafka pub/sub messaging system and then we describe our system architecture and model.

2.1 Apache Kafka

We chose Kafka [23], a popular, state-of-the-art, topic-based pub/sub system as our messaging system. In Kafka each published message corresponds to a specific topic and topics are further divided into partitions. The partitions are distributed across the brokers that comprise the Kafka cluster. Each partition can be hosted on a different broker, which denotes that a single topic can scale horizontally across multiple servers for redundancy and scalability.

A Kafka producer is an application that acts as a source of data in a Kafka cluster. A producer can publish messages to one or more Kafka topics; the message is directed to the appropriate partition. Incoming messages are assigned to *partitions* using a *consistent-hashing* mechanism on the message key [17] while partitions are assigned to the cluster's brokers using a round-robin policy. Each partition has exactly one partition leader which handles all the read/writes requests to that partition. If the replication factor is greater than one, the additional replica partitions act as followers. This synchronization is achieved through the ZooKeeper service. Kafka Consumers are subscribers wishing to read records from one or more topics and one or more partitions of a topic. Consumers can work together as part of a consumer group. Consumer groups act as a level of parallelism on a Kafka cluster as consumers that are part of the same group would be assigned with different partitions.

Despite the wide adoption of topic-based pub/sub systems, the problem of how to dynamically adjust the number of brokers in the cluster when overloads, load imbalances or skewness occurs due to the volume of messages in the pub/sub system, to maintain the service required by the users, still remains a significant challenge.

2.2 System Architecture and Model

Figure 1 presents our system architecture comprising a Kafka cluster running in a Docker Container where the containers communicate via an overlay network. The benefit of the Docker Container is that it simplifies and automates the

deployment process and seamlessly scale our system to use multiple machines. For a machine to be part of the system it needs to have a Docker daemon installed.

During operation, each Broker in the Kafka Cluster is configured to run with a JMX exporter where these metrics are exposed to a form that can be collected by the Prometheus monitoring tool (http://prometheus.io), an open source, metrics-based monitoring system. The collected statistics are sent to the Deep Reinforcement Learning Agent, which in turn makes the appropriate Scaling action. Three scaling actions are supported: *Scale Up*: a Broker needs to be added to the Kafka Cluster, *Scale Down*: a Broker must be removed from the Kafka Cluster, and *No Scaling*: no change in the number of Brokers in the Cluster.

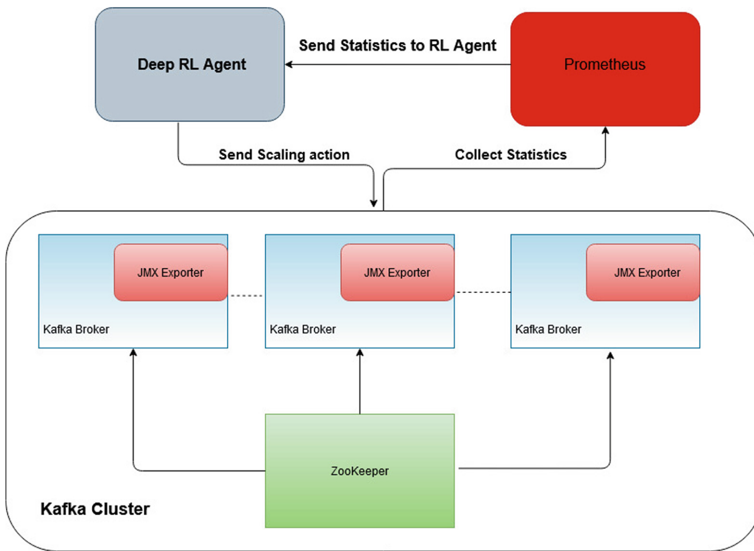

Fig. 1: System architecture

Our system collects a set of metrics from each Kafka broker via the use of Prometheus. Each Kafka Broker is configured to run with a JMX exporter where these metrics are exposed to a form Prometheus can retrieve. More specifically, our system keeps track of the following metrics: (i) *CPU utilization* (mean rate, 95th%), (ii) *number of bytes in/out per second* (mean rate, 95th%), (iii) *number of messages in/out per second* (mean rate, 95th%), (iv) *response queue time* (mean rate, 95th%) which denotes the time it takes to send the response to the Requestor, (v) *number of produce requests* (mean rate, 95th%) (vi) *requests per second* (mean rate, 95th%) (vii) *requests waiting in the purgatory* (mean rate, 95th%).

We denote as *total time* (mean rate, 95th%), the amount of time taken to service a request (Produce, Fetch Consumer and Fetch Follower), computed as

the sum of the following three types of requests: (a) *request queue time* (mean rate, 95th%): the time the request spends in the queue once it has been received but before processing starts, (b) *local time*: the time spent being processed by the partition leader, (c) *remote time*: The time spent waiting for the follower response before processing completes, We denote as *response queue time*, the time it takes to send the response to the requestor. The total time is utilized when we compute the reward that our agents collect.

Requests Waiting in the Purgatory. The purgatory holds requests waiting to be satisfied. It is only used for Produce and Fetch requests. Each type of request has different parameters that determine if it will be added to purgatory:

- Produce requests will be added to purgatory until the partition leader receives an acknowledgment from in-sync replicas. The number of acknowledgments the partition leader requires is determined by the **acks** parameter. When acks=all, means that the leader will wait all in-sync replicas to acknowledge the record and then send the next one.
- Fetch requests are added in the purgatory if there is not enough data to fulfill the request. So wait until enough data is available or the max waiting time for the request has passed.

Monitoring the size of purgatory is useful in order to determine the underlying causes of latency.

3 Proposed Methodology

3.1 Deep Reinforcement Learning

The goal of Reinforcement Learning (RL) is to discover which actions yield the highest numerical reward by trying them, not only for the immediate reward but also for all subsequent rewards. Reinforcement learning has important benefits compared to supervised learning in that it does not need labeled input/output pairs to be presented, and over unsupervised learning which is typically about finding hidden structure and correlations between a set of unlabeled data. One of the most popular RL algorithms is Q-learning. The goal is to learn a policy, which tells an agent what action to take under what circumstances. Q-Learning is a model-free algorithm as it does not require a model of the environment and the goal is to learn the value of an action in a particular state.

The core of the Reinforcement Learning algorithms is to estimate the action-value function, using the Bellman equation as an iterative update,

$$Q_{i+1} = E[r + \gamma \max_{a'} Q_i(s', a')|s, a]. \tag{1}$$

where r is the reward, γ is the discount factor and a is the learning rate. This kind of value iteration algorithms converge to the optimal action-value function as the number of iteration goes to infinity, $Q_i \rightarrow Q*$ as $i \rightarrow \infty$. These methods are impractical, because the action-value function is estimated separately for

each sequence, so we do not get any generalization. Also standard Reinforcement Learning is mostly limited to domains which are fully observed or to domains where features can be handcrafted e.g. with the bucket method. It works best when the number of possible states and actions are finite.

In our approach we use a non-linear function approximator i.e. a Neural Network. We refer to a Neural Network function approximator with weights θ as a Q-network. We train a Q-network by minimizing the loss function $L_i(\theta_i)$ at each iteration i,

$$L_i(\theta_i) = E_{s,a \sim \rho(.)}[(y_i - Q(s,a;\theta_i)^2] \tag{2}$$

where $y_i = E_{s' \sim Environment}[r + \gamma max_{a'} Q(s',a';\theta_{i-1})|s,a]$ is the target for iteration i and (s,a) is the probability distribution over states s and actions α. We refer to $\rho(.)$ as the behaviour distribution. Note that the parameters from the previous iteration θ_{i-1} are fixed when minimizing the loss function $L_i(\theta_i)$ and depend solely on the network weights, compared to targets used for supervised learning which are fixed before training.

If we differentiate the loss function with respect to the weights we have the following gradient:

$$\begin{aligned} \nabla_{\theta_i} = E_{s,a \sim \rho(.);s' \sim Environment}[(r + \gamma max_{a'} Q(s',a';\theta_{i-1}) \\ -Q(s,a;\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i)]. \end{aligned} \tag{3}$$

A closed-form solution to obtain directly the weights that minimize the loss function exists but may be time-consuming if we have a very large number of examples and features. So in our approach we use Stochastic gradient descent. If we update the Q-network at each time step we have the familiar Q-learning algorithm.

Deep Q Network Algorithm. Our first approach is based on a Deep Q Network. First we utilize a technique known as Experience Replay. We store the tuple (s,a,r,s') at each time step in a dataset D with maximum capacity N, known as replay memory. During the inner loop of the algorithm, we sample a mini-batch of experiences from the replay memory and train the Q-network. The sample is drawn uniformly from the replay memory i.e. every tuple has the save probability to be chosen. In our approach we sample experiences regardless of their significance. Alternatively, one could use the Prioritized Experience Replay method in order to replay important transitions more frequently, and therefore learn more efficiently. With the use of Experience Replay, we break the correlation of data, and because each experience is potentially used in many updates we have greater data efficiency. Overall, the use of the replay memory smooths the learning process and helps the Q-network to avoid to fall into local minimums and that we will eventually find the optimal policy of every state-action pair.

Deep learning requires a large amount of hand-labeled training data which are later used as the target values when minimizing the loss function. In our case the target value, for iteration i, has the following form:

$$y_i = E_{s' \sim Environment}[r + \gamma max_{a'} Q'(s',a';\theta_i)|s,a] \tag{4}$$

where Q' is a different Neural Network, called Target Network. The Target Network has the same architecture as the initial Q-network but with frozen parameters. Then, every C-steps we update the Weights of the Target Network to match the Weights of the initial Q-network. This leads to a more stable training because it keeps the target function fixed for C time steps.

Another benefit of the Deep Q Network algorithm is that it is able to find which input data play an important role on the behaviour of the Q-network and which are not. Given the above, we feed the Q-network with a 37 vector as input (this is the total number of metrics we collect from the Monitoring component), and it decides which inputs are important. So the best action to take here is to feed the network with all information available. Those inputs that are not significant will have weights approaching to zero.

Double Deep Q Network Algorithm. The Deep Q Network Algorithm is known to overestimate action values, which can impact training especially early on. If the DQN takes action 1 for example and learns a high Q value for that specific action, that means that action 1 is going to be selected more, compared to the other possible actions. This further overestimates the Q value for that state action which leads to training instability and poor performance.

The idea in the **Double Deep Q Network** to reduce overestimations is to decouple the selection and the evaluation of an action. As above, we use the Q-network to select and evaluate actions using the ϵ-greedy strategy. But when it comes to computing the target y_i, we first find which action to take using the Q-network Q and evaluate that action based on the target Q-network Q'. The formula for that is:

$$y_i = E_{s' \sim Environment}[r + \gamma Q'(s', argmax_{a'} Q(s', a'; \theta_i); \theta_i^-)|s, a] \tag{5}$$

As earlier, the weights of the target Q-network θ^- are replaced with the weights of the Q-network θ every C-steps.

3.2 Scaling Decisions

In this section we describe our process for adding and removing brokers from the Kafka cluster. When a broker is added, it will not automatically be assigned any partitions, so we have to come up with a reassignment plan to move already existing partitions to the newly added broker, in order to fully operate and be part of the Kafka cluster. For this purpose, we utilize a simple round robin partition technique. First, we calculate how many partitions must be moved to the new broker, the total number of partitions divided by the number of active brokers, including the new one. Then we start constructing our custom reassignment plan by simply beginning from a random broker, picking one partition from the broker and adding it to the reassignment plan which is in JSON format, and then sequentially perform the same operation on each broker we encounter, until we reach the appropriate amount of partitions. Using this scheme we can achieve a fair redistribution of the partitions among the available brokers in the Kafka cluster. So we are evenly spreading the load. When removing a broker from the

Kafka cluster, we follow the same procedure. First, we get the partitions which are located on the broker marked for removal. Then, we again construct a custom reassignment plan by allocating each partition to a broker following the round robin technique, starting from a random broker and sequentially going to the next one until all partitions are matched to a broker. We utilize the Kafka partition reassignment tool to move the partitions across the brokers.

This tool supports three modes:

- *generate*: In this mode, given a list of topics and a list of brokers, the tool generates a reassignment plan to move the topics to the brokers specified in the list.
- *execute*: In this mode, the user provides a reassignment plan and the tool executes it. It can be either the reassignment plan from the generate mode or a handcrafted one.
- *verify*: In this mode, the tool verifies if the partitions in the reassignment plan moved successfully to their specified broker.

We make use of the above modes provided by the partition reassignment tool. The generate mode outputs the current partition assignment, so we have a complete view of where each partition is located in the cluster. We use this information to construct our custom assignment. Note that we do not use the proposed assignment of the tool as it outputs random movements between the active brokers in the cluster and as a result incurs a large overhead. In our approach we tried to minimize the number of movements as much as possible. In the execute mode, we provide our custom reassignment plan as a json file. This is the step where the actual movement of the partitions takes place. Finally, the verify mode is important when we remove one of the brokers. We remove the broker from the cluster only when the reassignment plan finishes execution.

The operation of the Reassign Partitions Tool are summarized below.

1. First the tool updates the Zookeeper path "/admin/reassign_partitions" with the partition assignments we specified in the JSON file we created.
2. The Kafka Controller listens to the above path for changes. It gets notified from a ZooKeeper Watch.
3. For each specified partition, the following procedure is executed:
   (a) Start new replica partitions in RAR-AR (RAR = Reassigned Replicas, AR = original list of Assigned Replicas)
   (b) Wait until the new replica partitions are in sync with the leader partitions.
   (c) Check if the leader partitions are in RAR, if not elect a leader from RAR.
   (d) Stop replicas from AR-RAR.
   (e) Clear the Zookeeper path "/admin/reassign_partitions".

Note that the tool only updates the path. The Kafka Controller is responsible to execute the reassignment. To make sure that records are not lost during the cluster expansion, we utilize the acks setting in the Kafka producer configuration. It denotes the number of brokers that must receive the record before we consider the sending of a message successful. In our case we used the option, $acks = all$.

This way we ensure that we will not loose any records during cluster scaling, even if producers and consumer perform write and/or read operation on the partitions marked for movement.

4 Evaluation

We evaluated our approach in our local cluster comprising physical machines Intel Core i7-8700k and Intel Core i7-9750H, each with 16 GB DDR4 RAM, running Ubuntu 20.04 and with the Docker daemons installed. The training of the Deep Q Network and Double Deep Q Network was performed on an Nvidia RTX 2070 with 8 GB GDDR6 memory. The RTX features 2,304 CUDA cores which makes it ideal for performing multiple computations simultaneously compared to a single CPU, which in our case comprises 12 logical cores.

Our Deep Learning model is a 3-layer fully connected network. The first layer consists of 64 neurons, the second layer 128 neurons and the third one of 256 neurons. We initialize the weights of our network using Xavier uniform initializer, to address the problem of vanishing and exploding gradients [18]. One key point of our implementation is that we perform layer normalization on the input sample to improve the training speed. In our case, our input sample is a 37th dimensional vector with each column comprising a different feature. Each feature has its own range of values, e.g. one feature is the active number of brokers, at a given time point, ranging from 3 to 11 brokers, and another feature is the average producer latency, again at the same time point, ranging from 0 to 500 in milliseconds. So normalizing the input sample will help us reduce the training time.

Machine learning algorithms that use a variation of Gradient descent, as an optimization technique, require data to scale as it helps the model to converge quickly towards the global minimum. In standard Deep Learning, the most used normalization technique is batch normalization [19], but it cannot be applied on online tasks such as our Deep Q Network and Double Deep Q Network implementations. Thus, we used layer normalization [20], that was designed to overcome the drawbacks of batch normalization. Layer normalization can be applied on a single training sample compared to batch normalization, and that is the main reason we chose it for our network.

We utilize the following hyper-parameters: mini-batch size (number of training cases over which SGD update is computed) was set to 10, the replay memory size (SGD updates are sampled from the replay memory buffer) was set to 42, the target network update frequency was set to 5, the discount factor (gamma used in the Q-learning update) was set to 0.99, and the update frequency was set to 1 (selecting value 1 results in updating after every scaling action). The optimization method we used was RMSprop, a stochastic gradient descent method that maintains a moving (discounted) average of the square of gradients and divide the gradient by the root of this average. The learning rate used by the Adam optimizer was set to 0.00025, the initial value of the ϵ exploration was 1 and the final exploration was 0.1 and we perform 10 random action at the start of the

first two episodes. Some important aspects to note: (a) We do not clear out the replay memory after each episode, this enables us to recall and build batches of experiences from across episodes. (b) We used RMSprop as our optimization algorithm with zero momentum value(as in [21,22]).

Each experiment runs a total time of 30 min. At time-step 0 we create the required topics and initialize the producers and the consumers. We perform one scaling action every 40 sec. A scaling up or scaling down action takes roughly 20 sec. We performed experiments with 3 different reward functions, keeping the same network architecture and hyperparameters, and comparing the results of the two different learning algorithms, DQN and Double DQN. For the experiments we used a total of $1.8M$ records per producer, each record had a size of 124 bytes. We used 4 Producers, 4 Consumers. We varied the number of Brokers between 3 and 11 and each topic had 16 partitions. The DQN and Double DQN agents were trained for 10 episodes.

In our experiments we evaluated our approach with the following reward functions:

Reward Function as a Function of the Number of Active Brokers. In our first reward function we defined the immediate reward r as a function of the number of active brokers, as:

$$r_t = -NumberOfActiveBrokers_t \tag{6}$$

where we give a negative reward at each time step. This way we incentivize our DQN and Double DQN agents to reduce the number of brokers to the minimum.

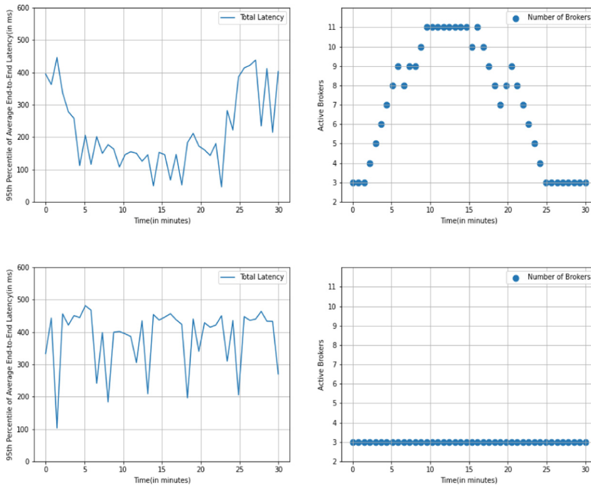

Fig. 2: DQN in early (top row) and later (bottom row) stages of training (reward as a function of brokers).

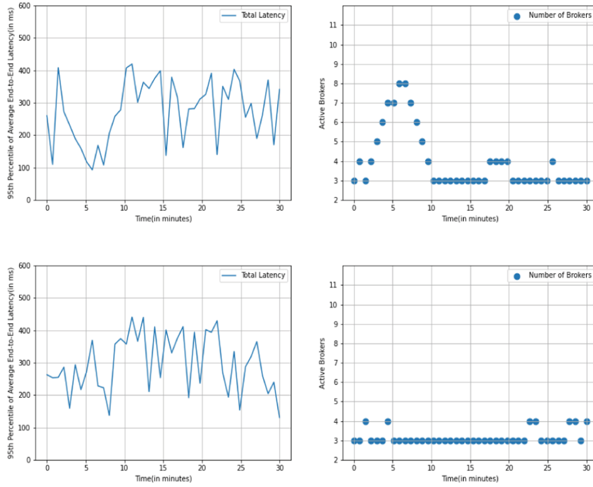

Fig. 3: Double DQN in early (top row) and later (bottom row) training (reward as a function of brokers).

Reward Function as a Service Level Objective (SLO). A service-level agreement (SLA) is a commitment between a service provider and a client. SLA defines the metrics by which the service is measured, and the penalties, in case the agreed-on service levels not be achieved. For our purposes we want the 95th% of the Average End-to End Latency to be under 200 milliseconds. Choosing the appropriate SLO was complex, but we set this value of 200 ms as it was reasonable for our setting with 11 active brokers. The reward in this case is a piecewise function and depends on the percentage difference of the actual value with the SLO.

Summarizing, our second reward function has the following form:

$$r_t = \begin{cases} -2C & \text{if Total Latency is 200\% over SLO} \\ -3/2C & \text{if Total Latency is 150\% over SLO} \\ -C & \text{if Total Latency is 100\% over SLO} \\ -1/4\,C & \text{if Total Latency is 50\% over SLO} \\ -1/10\,C & \text{if Total Latency is over 10\% SLO} \\ 0 & \text{Otherwise} \end{cases}$$

Reward Function as a Weighted Sum of Active Brokers and SLO. Our third reward function was defined with respect to both the number of active brokers and the SLO and had the following form:

$$NormalizedBrokerCost_t = -\frac{ABr - MinBr}{MaxBr - MinBr} \tag{7}$$

where AB is the number of Active Brokers at timestep t, MinBr is the minimum number of Brokers and MaxBr is the maximum number of Brokers. For our

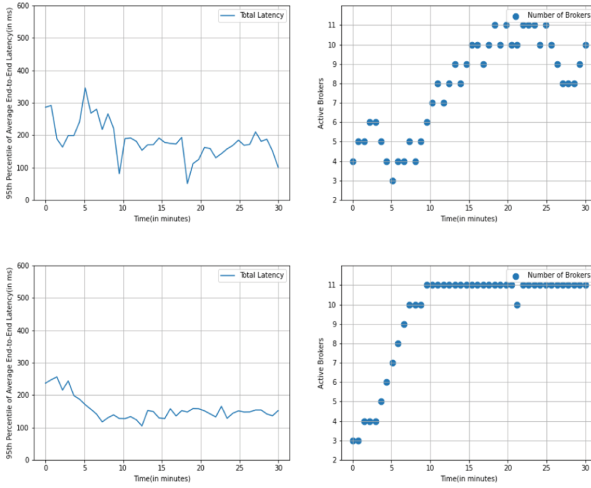

Fig. 4: DQN in early (top row) and later (bottom row) stages of training (reward as a function of SLO).

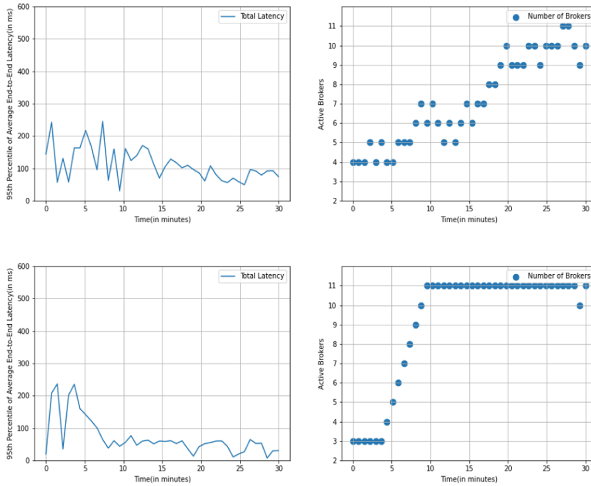

Fig. 5: Double DQN in early (top row) and later (bottom row) stages of training (reward as a function of SLO).

setup MinBr has a value of 3 while MaxBr a value of 11. The SLA cost has the following form:

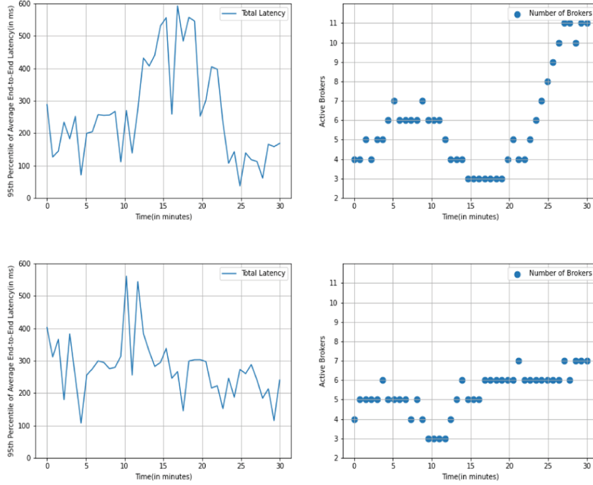

Fig. 6: DQN in early (top row) and later (bottom row) stages of training (reward as function of both active brokers and the SLO).

$$SLACost_t = \begin{cases} C & \text{TotalLatency is 200\% over SLO} \\ (80/100)C & \text{TotalLatency is 150\% over SLO} \\ (60/100)C & \text{TotalLatency is 100\% over SLO} \\ (40/100)C & \text{TotalLatency is 50\% over SLO} \\ (20/100)C & \text{TotalLatency is over SLO} \\ 0 & \text{Otherwise} \end{cases}$$

where C = −1. Note that SLAClost ranges between −1 and 0. We combine the different costs into a single cost function using a Simple Additive Weighting (SAW) technique. According to SAW, we define the reward function as follows:

$$r_t = W_{brokers} * NormalizedBrokerCost_t + W_{sla} * SlaCost_t \tag{8}$$

where $W_{brokers}$, W_{sla} satisfy the restriction $W_{brokers} + W_{sla} = 1$. In the experiments we used $W_{brokers} = W_{sla} = 50\%$ and the SLO was set to 300ms.

As we observe from the results presented above (Figs. 2, 3, 4, 5, 6 and 7), for the three different reward functions, both DQN and Double DQN agents performed well and quickly achieved the expected behavior, at each a scaling action. Our evaluation results illustrate that, independently of the reward function, each agent was able to achieve the goal i.e. to maximize the expected value of the cumulative sum of a received scalar signal (reward), with no adjustment of the Deep Learning Model or hyperparameters across each learning algorithm. Both agents performed equally well, with respect to the total number of rewards accumulated across each episode. The third reward function, Double DQN achieved higher performance on almost each training episode. Regarding the first reward

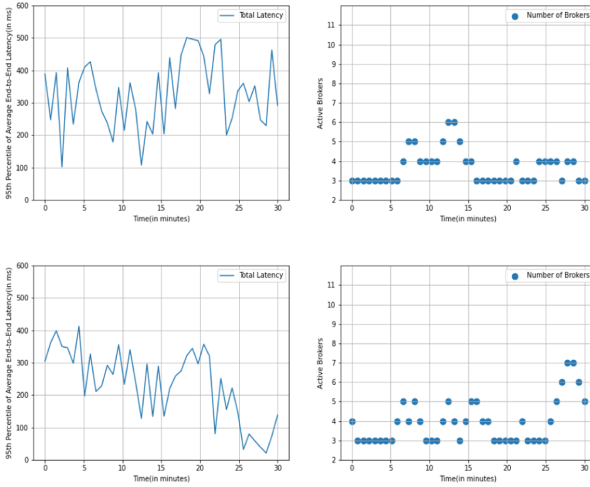

Fig. 7: Double DQN in (top row) and later (bottom row) stages of training (reward as function of active brokers and the SLO).

function, both agents learned that the optimal behavior was to keep the number of active brokers to the bare minimum. While on the second reward function, in order to keep the latency below the predefined threshold, both agents learned that in order to achieve that they must keep the number of active brokers to the maximum. When using the third reward function, there is a trade off between the number of active brokers and the SLO. As our experiments indicate, both agents managed to learn the optimal policy, satisfying the SLO most of the time while keeping the number of active brokers to the least required.

5 Related Work

There has been work on adaptation and load balancing techniques in pub/sub systems. The main difference between topic-based and content-based pub/sub systems is the fact that the latter organize brokers in an overlay network so the load balancing problem is solved by either dynamically changing the network topology [7] or by updating the routing paths [8]. The authors in [8] propose a new publication routing algorithm that takes into account broker resources and publication popularity among subscribers for content-based pub/sub systems. In [7] the authors propose a dynamic load balancing technique for content-based publish/subscribe systems. Their algorithm distributes the incoming load by offloading subscribers from heavily loaded brokers to less loaded brokers. However, they use simple threshold based techniques for determining when a node is overloaded and try to balance the load in the set of nodes that reside to the same overlay as they examine the problem in a content-based pub/sub system. In [9] the authors proposed an approach to address the problem of hot topics in

pub/sub systems by exchanging load related information about the brokers. The authors in [10] examined the load balancing problem in topic-based pub/sub systems where the goal is to automatically construct an overlay network and then establish the appropriate publication routing. Similarly, authors in [11] proposed a distributed algorithm to build and maintain a routing structure that can be used by topic-based pub/sub systems that exploit overlay networks. Furthermore, in [12] they deal with the problem of congestion avoidance in content-based pub/sub systems. They consider the congestion invoked due to unsubscriptions. In a topic based system like Kafka this would not create problems as there is no hierarchy (*i.e.,* similar to the routing tables) of the brokers. Similarly, Pietzuch et al. [13] presented a pub/sub congestion control scheme that adjusts the rate of publishing new messages, allowing brokers under recovery to eventually catch up, and other brokers to keep up. In our problem we assume that we cannot control the rate at which publishers send their messages to the brokers. In our work our aim is to use machine learning techniques with system performance metrics in order to build the appropriate robustness and drive the decisions that will enable us to develop elastic and scalable pub/sub systems.

Machine learning techniques provide a promising adaptation approach to maintaining QoS properties of QoS-enabled pub/sub middleware in dynamic environments. The problem of autonomic adaption has also been studied in the context of service level agreements. For example, Herssens et al. [14] study the problem of autonomically adapting service level agreements (SLAs) when the context of the specified service changes, to offer QoS for Web services. Their goal is to negotiate the QoS agreement to fit the needs of the dynamic environment. [15] et al. and [16] et al. apply machine learning techniques to deal with parameters uncertainty or simplify the configuration of QoS-enabled middleware and adaptive transport protocols to maintain specified QoS as systems change dynamically. The results of their work show that decision trees and neural networks can effectively classify the best protocols to use in adaptive environments.

6 Conclusions

In this paper we propose our approach for building a robust, scalable and elastic pub/sub system utilizing Deep Reinforcement Learning techniques. We evaluated our approach on Apache Kafka, with two algorithms, Deep Q Networks and Double Deep Q Networks, with three different reward functions. Our performance evaluation illustrated that, each agent was able to achieve the goal and thus can be efficiently utilized to address system changes or overload situations.

Acknowledgment. This research has been supported by the H2020 LAMBDA Project 734242, the EU ICT-48 2020 project TAILOR (No. 952215), the H2020 AutoFair project (No. 101070568).

References

1. César, C., Zhang, K., Kemme, B., Kienzle, J., Jacobsen, H.A.: Publish/subscribe network designs for multiplayer games. In: Proceedings of the 15th International Middleware Conference, pp. 241–252 (2014)
2. Yoav, T., Naaman, N., Harpaz, A., Gershinsky, G.: Hierarchical clustering of message flows in a multicast data dissemination system. In: IASTED PDCS, Phoenix, AZ, USA, vol. 5 (2005)
3. Sharma, Y., et al.: Wormhole: reliable pub-sub to support geo-replicated internet services. In: 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015, vol. 15, Oakland, CA, USA, pp. 351–366 (2015)
4. Google Cloud Pub/Sub. https://cloud.google.com/pubsub/
5. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: a distributed messaging system for log processing. In: Proceedings of the NetDB, Athens, Greece, pp. 1–7, June 2011
6. Apache Pulsar. https://pulsar.apache.org/
7. Cheung, A.K.Y., Jacobsen, H.-A.: Dynamic load balancing in distributed content-based publish/subscribe. In: van Steen, M., Henning, M. (eds.) Middleware 2006. LNCS, vol. 4290, pp. 141–161. Springer, Heidelberg (2006). https://doi.org/10.1007/11925071_8
8. Salehi, P., Zhang, K., Jacobsen, H.A.: PopSub: improving resource utilization in distributed content-based publish/subscribe systems. In: Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, pp. 88–99 (2017)
9. Dedousis, D., Zacheilas, N., Kalogeraki, V.: On the fly load balancing to address hot topics in topic-based pub/sub systems. In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), ICDCS 2018, Vienna, Austria (2018)
10. Chen, C., Jacobsen, H.-A., Vitenberg, R.: Algorithms based on divide and conquer for topic-based publish/subscribe overlay design. IEEE/ACM Trans. Networking **24**(1), 422–436 (2016)
11. Turau, V., Siegemund, G.: Scalable routing for topic-based publish/subscribe systems under fluctuations. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), ICDCS 2017, Atlanta, GA, USA, pp. 1608–1617. IEEE (2017)
12. Chen, M., Hu, S., Muthusamy, V., Jacobsen, H.A.: Congestion avoidance with incremental filter aggregation in content-based routing networks. In: 2015 IEEE 35th International Conference on Distributed Computing Systems, ICDCS 2015, Columbus, OH, USA, pp. 557-568 (2015)
13. Pietzuch, P.R., Bhola, S.: Congestion control in a reliable scalable message-oriented middleware. In: Endler, M., Schmidt, D. (eds.) Middleware 2003. LNCS, vol. 2672, pp. 202–221. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44892-6_11
14. Herssens, C., Faulkner, S., Jureta, I.J.: Context-driven autonomic adaptation of SLA. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 362–377. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89652-4_28
15. Hoffert, J., Mack, D., Schmidt, D.: Using machine learning to maintain pub/sub system GOS in dynamic environments. In: Proceedings of the 8th International Workshop on Adaptive and Reflective Middleware, ARM 2009, Urbana, IL, USA, pp. 1–6 (2009)

16. Russo, G.R., Cardellini, V., Presti, F.L.: Reinforcement learning based policies for elastic stream processing on heterogeneous resources. In: 13th ACM International Conference on Distributed and Event-based Systems (DEBS 2019), pp. 31–42, Darmstadt, Germany, June 2019
17. Karger, D., et al.: Web caching with consistent hashing. Comput. Netw. **31**(11), 1203–1213 (1999)
18. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, vol. 9, pp. 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010
19. Ioffe, S., Szegedy, C.: Batch normalization: accelerating deep network training by reducing internal covariate shift (2015)
20. Ba, J.L., Kiros, J.R., Hinton, G.E.: Layer normalization (2016)
21. Mnih, V.: Playing Atari with deep reinforcement learning, NIPS deep learning workshop (2013)
22. Mnih, V., et al.: Human-level control through deep reinforcement learning nature, vol. 518, pp. 529–33, February 2015
23. Kafka streams API. https://kafka.apache.org/documentation/streams/