Journal of Network and Computer Applications 190 (2021) 103143

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Optimizing the resource usage of actor-based systems

Hai T. Nguyen[a], Tien V. Do[a,*], Csaba Rotter[b]

[a] Analysis, Design and Development of ICT systems (AddICT) Laboratory, Department of Networked Systems and Services, Budapest University of Technology and Economics, H-1117, Magyar tudósok körútja 2., Budapest, Hungary
[b] Nokia Bell Labs, H-1083 Budapest, Bókay János utca 36-42, Budapest, Hungary

ARTICLE INFO

Keywords:
IoT
Actor
Resource management
Reinforcement Learning

ABSTRACT

Runtime environments for IoT data processing systems based on the actor model often apply a thread pool to serve data streams. In this paper, we propose an approach based on Reinforcement Learning (RL) to find a trade-off between the resource (thread pool in server machines) usage and the quality of service for data streams. We compare our approach and the Thread Pool Executor of Akka, an open-source software toolkit. Simulation results show that our approach outperforms `ThreadPoolExecutor` with the timeout rule when the thread start times are not negligible. Furthermore, the tuning of our approach is not tedious as the application of the timeout rule requires.

1. Introduction

In the past few years, we have experienced a significant rise in networking devices and the global network infrastructure behind them called the Internet of Things (IoT). Gartner (2017) predicted that the number of IoT units would exceed twenty billion and hardware spending from the consumer and the business segment together would reach three trillion USD. There are many use cases regarding the IoT technology: RFID tags can be attached to products to track their origin; data through vehicle-to-vehicle and vehicle-to-infrastructure communications can be utilized to avoid congestion and reduce incidents; sensors could monitor the state of a vehicle, the health of a specific person, or other environmental parameters (Bandyopadhyay and Sen, 2011; Xu et al., 2014; Albahri et al., 2021).

Systems that serve and process data from IoT devices are often implemented with the actor model programming paradigm (Diaz Sánchez et al., 2015; Haubenwaller and Vandikas, 2015). The Akka toolkit is a framework built on the JVM that supports the development of actor-based systems (Roestenburg et al., 2015; Bonér, 2010). Actors are designed for distributed computing and can handle highly concurrent IoT data streams. Also, the hierarchical structure of an actor-based system can be directly mapped into the relationships between IoT groups and devices. Actors in Akka share a set of threads in a thread pool, and the performance of the IoT application heavily depends on the size of the thread pool. A common way to set a static thread pool size is by trial-and-error. The system operator either increases or decreases the thread count based on measurement results. In a dynamic environment, e.g. under varying traffic, Akka allows the thread pool to change its size dynamically. We looked into the timeout rule used by Akka's thread pool managing entity. In this rule, the operator may set a timeout parameter to adjust the system to varying traffic. However, finding this value requires expert knowledge and may need a tedious trial-and-error process.

Motivated by the need for a good resource management solution, we propose an approach based on reinforcement learning (RL) for controlling the thread pool size to minimize resource usage while maintaining a certain QoS level. Our contributions are as follows:

- We defined a threshold-based multi-objective reward for the reinforcement learning problem. It considers the service waiting times for Qos and the thread pool size for efficiency.
- We used population-based training (Jaderberg et al., 2017) and grid search in combination to find the hyperparameters for the RL algorithm. These hyperparameters influence how close we could get to optimal operation and the stability of the algorithm.
- We compared the performance of the RL algorithm with the timeout rule used by the Akka actor framework. Through simulations, we showed that when threads start instantaneously, the RL algorithm performs similarly to the timeout rule. When the thread start time is not negligible, the RL algorithm performs better than the timeout rule.

The rest of the paper is organized as follows. In Section 2 we overview the related literature. In Section 3 we describe a scenario in which an actor based system is built for an IoT use case. In Section 4 we present our approach. In Section 5 we review the RL algorithm we used

* Corresponding author.
E-mail address: do@hit.bme.hu (T.V. Do).

https://doi.org/10.1016/j.jnca.2021.103143
Received 21 January 2021; Received in revised form 26 May 2021; Accepted 7 June 2021
Available online 16 June 2021

in our experiments. In Section 6 our experiment results are discussed. Finally, Section 7 contains our conclusions and future directions.

2. Related works

2.1. Resource management in for IoT

The problem of optimizing resources has always been central for many. Lately, resource management involving IoT has gained more focus. Musaddiq et al. (2018) investigated the resource managing capabilities of various operating systems in IoT. Zhang et al. (2017) studied the resource management between network slices in a 5G system, where IoT applications would get a separate slice. Alazab et al. (2021) used an enhanced rider optimization algorithm to find the optimal head nodes in IoT clusters and Alazab et al. (2020) used long short-term memory (LSTM) to predict resource usage in smart grids.

Furthermore, numerous studies have been published on cloud or fog computing resource management with IoT applications in mind. Aazam and Huh (2015) devised a dynamic pricing algorithm based on resource usage estimation in a fog computing system. Skarlat et al. (2016) constructed a framework with fog cells and fog colonies for IoT and created a resource provisioning algorithm for the framework. Barcelo et al. (2016) formulated the service distribution problem in IoT clouds and provided a solution through linear programming.

2.2. Thread pools

Approaches to control threads can be classified into the reactive and the proactive policy (Singh et al., 2019). The reactive policy takes actions to respond to the environmental changes, whereas the proactive approach predicts these changes and adjusts the number of threads accordingly.

In Ogasawara (2008), each service's load is continuously monitored, and thread counts of various thread pools were adapted dynamically according to load. The adaptive algorithm moves idle threads from thread pools with a low load to thread pools with a higher load. Kang et al. (2008) used the average exponential scheme to predict the number of required working threads and started other threads if necessary. Threads are stopped after a certain timeout period. Chen and Lin (2010) used the M/M/c/K/∞/FCFS queuing model to estimate mean waiting times and queue lengths to adjust the thread pool size in a middleware system dynamically. Lee et al. (2011) used a trendy exponential moving average scheme to predict and set the worker thread count and used the M/M/c queuing model to find an adequate minimum thread pool size. García-Valls (2016) identified the thread pool as one of the critical resources to manage in middleware applications. The author proposed an interception layer between the middleware and the operating system to control the upper and lower bounds of the thread pool's size.

2.3. Reinforcement learning for resource management

Machine learning (ML), or more specifically, reinforcement learning (RL), could provide an excellent alternative to managing resources by hand. RL is a technique for solving Markov Decision Processes (MDPs) (Sutton and Barto, 2018). In RL, an agent interacts with an environment described by the MDP, and over time it learns the optimal actions to pick to maximize the long term reward. RL algorithms like Q-learning (Watkins and Dayan, 1992) or policy gradient methods (Sutton et al., 2000) have been around for decades. However, only in the past few years have they received more use and attention due to hardware power developments that allowed computationally more demanding function approximation techniques like deep neural networks (DNN). When combined with DNN, RL is also referred to as deep reinforcement learning (DRL). Recently various feats have been achieved with DRL, such as machines learning to play ATARI games (Mnih et al., 2015) or Go (Silver et al., 2016). DRL has also found numerous practical applications like power consumption management (Tesauro et al., 2008), resource management (Mao et al., 2016), or robotics (Levine et al., 2018).

Cheng et al. (2018) investigated a deep RL based resource provisioning and task scheduling algorithm for large scale cloud service providers. Their goal was to minimize energy consumption using deep Q-networks (DQN) with two stages of action selection. They showed that the RL algorithms could outperform heuristic baseline methods. Wei et al. (2019) used Q-learning to rent on-demand and reserved virtual machines dynamically for a SaaS provider. Bibal Benifa and Dejey (2019) created an auto-scaling algorithm based on SARSA for clouds.

Jin et al. (2018) used fuzzy Q-learning to control a cloud-based web application. They defined the system's state using the workload and the virtual machine count and used the long term profit as the reward. They carried out experiments with simulations and in a real testbed. Jin et al. (2019) argued that exploiting some domain knowledge might be a better approach than tackling the problem in an entirely model-free manner.

Arteaga et al. (2017) dealt with scaling network functions. They combined Q-learning with a Gaussian process to estimate the system's future states to avoid executing erroneous actions. Mao et al. (2019b) introduced a new baseline to reduce the variance in environments with random input. They used their method on a load balancing task and a bitrate adaptation problem. Mao et al. (2019a) summarized the RL problems related to computer systems and created a framework for training RL algorithms for these problems.

Tahsien et al. (2020) provided a review of ML approaches for the security of IoT against different types of possible attacks. They also identified some research challenges on the application of ML to the security of IoT systems. Chowdhury et al. (2019) proposed a drift adaptive deep RL algorithm for scheduling and allocating resources in an IoT application. However, the performance of an IoT application is approximated with the simple M/M/1 queue.

To our best of knowledge, there have been no works that dealt with the application of RL to manage the resource of Actor-based systems.

3. Problem description

3.1. An implementation of an IoT application using the actor model

Akka is a free and open-source toolkit for the development of distributed applications (Roestenburg et al., 2015; Bonér, 2010). It uses the computational actor model, in which the basic building blocks are called the actors (Hewitt, 2010). An application that connects to IoT devices sets up and manages sessions and processes the sensor data in a server would employ actors.

Actors communicate through messages only. Upon receiving a message, an actor may either send a finite number of messages to other actors with known addresses; spawn new actors; or decide how to handle the following message. The actor may execute any amount of these actions in any order. Furthermore, these actions may be carried out in an asynchronous manner enabling a highly concurrent system. The actor model makes building distributed systems more convenient for the developer as it hides away lower-level elements like threads or locks. Note, though, that these low-level objects may significantly impact the IoT system's performance.

The main focus of Akka is to support actor-based concurrency. It is worth emphasizing that Akka can be used to build IoT platforms to process data from IoT devices (for example, ThingsBoard). An IoT platform's job is to provide a safe and secure connection to different IoT devices and lay the groundwork for applications and analytics. Fig. 1 outlines a simplified architecture of an IoT platform similar to the one used by ThingsBoard. We can see that IoT devices send data from the outside world through the Message Queuing Telemetry Transport (MQTT) protocol. If there is no connection established yet,

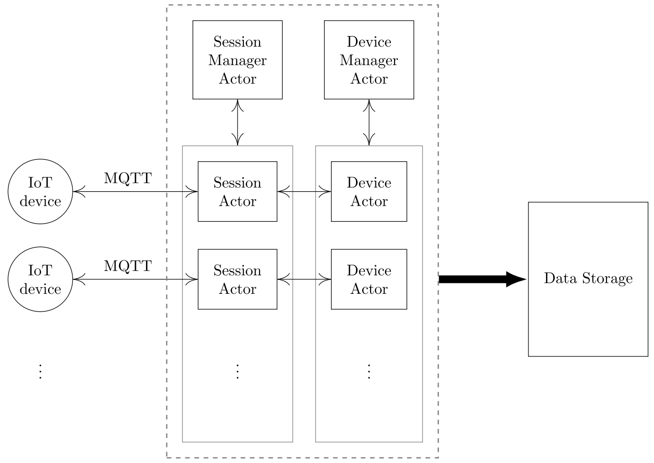

Fig. 1. IoT platform.

the Session Manager Actor creates a Session Actor to manage the IoT device session. Meanwhile, the Device Manager Actor makes a Device Actor to process the incoming data stored in persistence for further analytics.

3.2. Threads in Akka

Akka provides a runtime environment for the execution of distributed applications on the Java Virtual Machine (Akka documentation). The management of resources, i.e. the mapping of actors and system-level threads (operating system's resources), is based on the solution of the `ThreadPoolExecutor` of Java (Java TM Platform).

In Akka, the coordination of processing the messages is done by the dispatcher (Akka documentation). If an actor has a message in its message queue, the dispatcher may take that actor and assign it a thread to process that message. Threads are then mapped to CPU cores by the scheduler in the operating system. Fig. 2 shows us the relationship between actors, the thread pool and the CPU. Note that some actors may need to wait for available threads if there are more actors than threads. Similarly, if there are more threads than the number of CPU cores, the cores will be shared between the threads by the scheduler.

The assignment and the execution of the actor's code are done through the `ExecutorService`. In Akka, the two most common executors are the `ForkJoinExecutor` and the `ThreadPoolExecutor`. The former is the default executor and utilizes a static, fixed-sized thread pool, which the developer can configure manually. The latter can scale the thread pool dynamically depending on the system's demand. Since Akka is written in Scala and runs on the JVM, these executors are identical to those described in Java-related literature.

By default, the `ThreadPoolExecutor` uses the timeout rule as a policy (Java TM Platform). In this rule, threads are started on-demand. If a new task arrives and all other threads are busy, a new thread is created in the thread pool to process it. After finishing the task, the thread becomes idle until there is another task to be processed. If the thread remains inactive for a specific amount of time called the timeout (or keep-alive time), the thread is terminated to free up capacity. This allows the system to scale down the pool size when the traffic is low and scale up if traffic is high. The default timeout period set by Akka is 60 s.

Table 1
Notations used to describe the thread pool environment.

λ_{session}	Arrival rate of sessions
$X(t)$	Number of sessions
$Y(t)$	Number of threads
$B(t)$	Number of busy threads
L	Maximum number of sessions
N	Maximum number of threads
τ_{arrive}	Interarrival time of packets
η	Mean length of a session
τ_{service}	Base service time of a packet
t_{service}	Real service time of a packet
t_{start}	Thread start time
y_{lim}	Number of CPU cores
M	Capacity of an actor's message queue

3.3. A need to control the number of threads depending on traffic load

When a new IoT device connects to the system, an actor is created to open and manage a session, and another actor processes the data coming from the device. We will only consider data processing actors and assume that the thread and CPU time every other type of actors need is negligible.

Theoretically, there is no limit on how many actors we can spawn to execute these tasks simultaneously. However, the performance highly depends on the size of the dispatcher's thread pool beneath the actors. If there are not enough threads, the actors might end up waiting for each other. Unfortunately, we cannot just increase the thread pool size endlessly because only one thread can run on a CPU core, and a high number of threads would increase the time of context switching. Furthermore, idle threads may also use up resources. Therefore it is crucial to keep their number as small as possible. One of the developer's challenges is finding the optimal number of threads for a given application.

Suppose sessions arrive randomly with rate λ_{session}. We will denote the number of sessions at time t with $X(t)$. The value of $X(t)$ is limited by the maximum number of sessions in the system L, that is $0 \leq X(t) \leq L$, where L depends on the capacity of the physical machines. During the session, the IoT data arrive through packets with τ_{arrive} fixed interarrival times, which the actor then processes. We assume the length of a session is also random with mean η. When the IoT device disconnects, the session gets closed, and the related actors get removed from the system. Note that the processing of a packet may be finished after its session is expired, and the length of a session does not necessarily equal the actor's lifetime.

Let $Y(t)$ denote the number of threads at time t. The actors share these threads to process their packets. A thread may be in one of the following three states: *booting*, if it was started but has not finished the initialization process yet; *busy*, if it is processing an actor's message; or *idle*, if it is initialized, but is currently not working. Let $B(t)$ denote the number of busy threads at time t.

In this paper, to show the applicability of the RL technique, we assume that packets have the same size and that the service time of the packets is a fix value t_{service} until the number of busy threads B reaches a limit y_{lim} after which the service time of the tasks increases due to the threads resorting to sharing CPU resources. In practice, y_{lim} is the number of CPU cores itself. Fig. 2 shows us a system with X actors, Y threads, and y_{lim} CPU cores. We can see that an actor is assigned to a thread if there is an available thread in the pool. A thread can be assigned to a CPU core to execute its actor's task.

Threads are assigned to the actors in a round-robin manner by the dispatcher. This means that actors are ordered, and thread assignments follow this order. After a thread finishes its job, the dispatcher assigns it to the next waiting actor to process its packet. If no actors are waiting, the thread becomes idle until the dispatcher assigns it again to an actor. If all threads are busy upon the arrival of a packet, the packet is placed into the actor's message queue. We assume a limit of M to

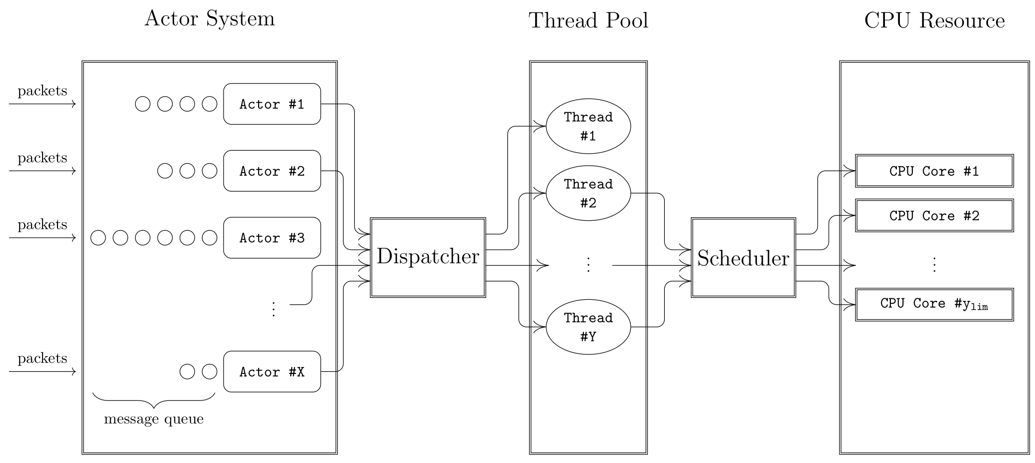

Fig. 2. The dispatchers in Akka assigns threads to the Akka actors. The mapping between the threads and the CPU cores is done by the operating system's scheduler.

Table 2
Notations used for describing the MDP.

$\mathcal{S}$	Set of states in the MDP
$\mathcal{A}$	Set of actions in the MDP
r	Reward function in the MDP
γ	Discount factor in the MDP
T_i	Decision times ($i = 0, 1, \ldots$)
ΔT	Time gap between two decisions
π	Policy function
V^{π}	Value function following policy π
s	State
s_i	State at time T_i
a_i	Action at time T_i
r_i	Reward at time T_i
$\hat{\lambda}$	Approximated arrival rate
$\hat{\delta}_i$	Approximated mean waiting time
$\hat{p}_{b,i}$	Approximated blocking rate
δ_{th}	QoS threshold for the mean waiting time
$p_{b,th}$	QoS threshold for the blocking rate
κ	Reward multiplier hyperparameter

the number of packets waiting in the message queues, which means that new incoming packets get thrown away if the number of waiting packets in the system is M upon their arrival.

The number of threads in the thread pool may be changed dynamically through the dispatcher. This thread count is upper bounded by N which can be set in a configuration file and depends on the capabilities of the physical machine, that is, $Y(t) \leq N$. We assume termination of threads is instantaneous, but turning them on requires t_{start} time. To satisfy specific QoS requirements, we need to have a sufficient amount of threads in the pool.

Using the `ThreadPoolExecutor` the operator may tweak the timeout value to adjust the thread pool size indirectly. This also alters the trade-off between QoS and performance. High timeouts result in a large thread count and better QoS, whereas a lower timeout comes with a smaller thread count and degrades QoS. Finding the correct value may be a tedious task for the operator, and this process may be repeated every time QoS requirements change. Furthermore, if t_{start} is high, we need higher timeouts too. Otherwise, turning off threads would result in high waiting times.

Our goal is to find a more efficient control of the thread pool size. Unfortunately, we cannot assume that the traffic is constant, which means that the optimal number for thread count may change throughout the system's lifetime.

For the list of notations describing the thread pool, see Table 1.

4. A formulation of a problem

To apply RL, first, we formalize the problem as a Markov Decision Process (MDP) with a 5-tuple $\langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$, where $\mathcal{S}$ is the set of states in the system; $\mathcal{A}$ is the set of actions; $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$ describes the transition probabilities between states; $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ describes the immediate reward between state transitions; and $\gamma \in [0,1]$ is the discount factor.

In the MDP framework, an agent interacts with the system. At timestamp T_i ($i = 0, 1, \ldots$) the agent observes the state of the environment and decides on executing an action. As a result, the agent receives reward feedback from a system. In our case observations are made periodically with period ΔT, that is $T_{i+1} - T_i = \Delta T$.

At each timestep, the agent decides according to a policy $\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A})$ which is a mapping between the set of states $\mathcal{S}$ and a probability distribution $\mathcal{P}$ over the set of actions $\mathcal{A}$. The agent's goal is to find the optimal policy that maximizes the long term expected reward when followed. One of the main advantages of RL lies in its model-free property, which means that it is not expected of the RL agent to possess knowledge of the environment. The agent does not require the p transition probability function to reach the optimal policy.

One of the most critical steps in solving MDPs with RL is defining the state space, the action space, and the reward function. The description of a state has to include every information that can be essential for making a decision. However, it also needs to be compact so that the RL agent could explore the state space in a reasonable time. Similarly, having a smaller action space can make it easier for the RL agent to learn, but it also limits the possible moves it can choose. Lastly, the reward function drives the RL agent into the desired operating point when optimized. Usually, this is a difficult task, and a poorly defined reward may result in the RL agent being stuck in unwanted states.

4.1. State space

The observation of the agent is compiled into a state representation $s \in \mathcal{S}$. The definition of $\mathcal{S}$ is crucial for the RL since including too many variables would explode the state space. It would unnecessarily elongate the training process, whereas too few variables may make it impossible for the agent to differentiate otherwise distinct states.

At any time t the state can be represented by a 4-tuple $s(t) = \left\{X(t), Y(t), B(t), \hat{\lambda}(t)\right\}$, where $X(t)$ is the number of active sessions, $Y(t)$ is the number of threads (including *booting*, *busy*, and *idle* threads), $B(t)$ is the number of busy threads in the thread pool, and $\hat{\lambda}(t)$ is the measured arrival rate during the previous ΔT time period. Note that the upper limit for the number of sessions $X(t)$ and the number of threads $Y(t)$ are L and N respectively, that is $0 \leq X(t) \leq L$, $0 \leq Y(t) \leq N$ and $0 \leq B(t) \leq Y(t)$. We will denote the states observed by the agent with s_i ($i = 0, 1, 2, \ldots$), where $s_i = s(t = T_i)$.

4.2. Action space

At each decision point T_i the agent observes a state s_i, from which it needs to take an action a_i with probability $\pi(a_i|s_i)$. $\mathcal{A}$ describes the set of actions that the agent may decide on. Note that the state of the system may change immediately after the execution of action a_i, but the new state is not necessarily identical with s_{i+1}, as s_{i+1} is the state that would be observed at time T_{i+1}. We considered two cases for the set of actions (we will denote them with $\mathcal{A}_1$ and $\mathcal{A}_2$).

4.2.1. On/off actions

In the first scenario, we consider only actions that either turn on or turn off one thread. Let us define the following actions:

- `START_THREAD`: a new thread is initialized in the thread pool;
- `STOP_THREAD`: an idle thread in the thread pool is stopped, if all threads are busy working, then the first to finish its job will be stopped;
- `NoOp`: no change is made.

That is $\mathcal{A}_1 = \{\texttt{START\_THREAD}, \texttt{STOP\_THREAD}, \texttt{NoOp}\}$. Obviously not all three actions are possible in every state. For action selection the following rules apply:

- if $Y(T_i) = N$, we cannot start any more threads, which means that `START_THREAD` cannot be selected;
- if $Y(T_i) = 0$, there are no threads to stop, that is `STOP_THREAD` cannot be selected.

4.2.2. Multi-actions

The advantage in using $\mathcal{A}_1$ is that it is a small action space, which is easy to explore. Thus the RL algorithm can learn and converge faster. However, it leaves little freedom to the operator since it can only react to changes by either stopping or starting precisely one thread. Therefore, we experimented with another, larger action space, where multiple threads could be turned on or off at once.

In the second case we define only one type of action, $\mathcal{A}_2(T_i) = \{B(T_i), B(T_i)+1, \ldots, N\}$, where $B(T_i)$ is the number of busy threads at decision time T_i. In this setting the agent could select any value between $B(T_i)$ and N and start or terminate multiple threads in order to set the thread pool size to that value. The action $a_i \in \mathcal{A}_2(T_i)$, at time T_i would set $Y(T_{i+1}) = a_i$ and would be the thread count for the time period $(T_i, T_{i+1}]$. Obviously this is a much larger action space than $\mathcal{A}_1$, but in this case the agent has more flexibility in making adjustment in the system.

Note that the name *multi-action* scenario refers to the fact that we can start or terminate multiple threads in one action and not that we may execute multiple actions at a decision point T_i.

4.3. Reward function

The reward function $r(t)$ describes the RL agent's objective to optimize. When the system reaches a decision point at time T_i, the agent receives the immediate reward $r_i = r(T_i)$ which is calculated based on the performance measures collected between the previous and the current decision points.

A well-calibrated thread pool should minimize resource usage while maintaining the quality of service (QoS). Here we are facing a multi-objective optimization problem as we want to minimize the number of threads in the thread pool while also adhering to the constraint of keeping the mean waiting time of the packets below a threshold level. A common way to tackle this is to aggregate the components into a single real reward value. This can be a very challenging task as different components may have different measures and numerical ranges. For more on multi-objective reinforcement learning see (Liu et al., 2015).

Motivation for a threshold-based objective for the RL algorithm is the provisioning of QoS for applications, i.e., the processing time requirement (QoS requirement) of a message carrying IoT data. The rationale behind the QoS provisioning is that processing data sensors that monitor critical events (like fire, …) needs a short time, while processing data of a greenhouse may not be so critical. We used the following reward function inspired by threshold-based algorithms:

Table 3
Notations used for the learning algorithms.

`tr_st`	Number of training steps
θ	Policy neural network parameter vector
ω	Value neural network parameter vector
$\bar{r}$	Mean reward
k	Epoch
b	Batch size
$\hat{A}^{\mathrm{GAE}}$	Generalized advantage estimator
χ	Generalized advantage estimation parameter
H	Entropy function
ξ	Entropy multiplier coefficient
α_θ	Policy function learning rate
α_ω	Value function learning rate
α_R	Reward averaging factor
ϵ	Clipping ratio
`par`	Degree of parallelism
l	Population based training steps
m	Population based training repetitions
c_κ	Reward multiplier perturbing factor
c_ξ	Entropy multiplier perturbing factor
ϵ_κ	Reward multiplier perturbing range
ϵ_ξ	Entropy multiplier perturbing range
n	Grid search run count

$$r_i = \begin{cases} -\kappa\hat{\delta}_i & \text{if } \delta_{th} < \hat{\delta}_i \text{ or } p_{b,th} < \hat{p}_{b,i} \\ -Y(T_i) & \text{if } \delta_{th} \geq \hat{\delta}_i \text{ and } p_{b,th} \geq \hat{p}_{b,i}, \end{cases} \tag{1}$$

where

- $\hat{\delta}_i$ and $\hat{p}_{b,i}$ are the mean waiting time and blocking rate of packets measured between the previous and the current decision points T_i and T_{i+1}, $i \in \mathbb{N}$,
- δ_{th} and $p_{b,th}$ are the threshold values determined by the QoS,
- parameter κ is simply a multiplier to solve the issue of $|\hat{\delta}_i| \ll |Y(T_i)|$ when $\hat{\delta}_i \approx \delta_{th}$. Without κ the $Y(T_i)$ value would numerically outweigh $\hat{\delta}_i$ in the expectation of the reward when the waiting time is close to δ_{th}.

The intuition behind reward function (1) is that if $\hat{\delta}_i$ or $\hat{p}_{b,i}$ is too high and higher than the threshold value, then we need to minimize $\hat{\delta}_i$ which is the same as maximizing $-\hat{\delta}_i$. However, if $\hat{\delta}_i$ and $\hat{p}_{b,i}$ are low enough and do not exceed the threshold δ_{th} and $p_{b,th}$, then we should minimize the number of threads to save resources. This is done by maximizing $-Y$. Note that the reward having a negative value can be interpreted as a cost.

We also experimented with other possible reward functions, like Kleinrock's power function (Kleinrock, 2018), but this objective cannot be adjusted for different QoS requirements. We tried a simple weighted sum of rewards too, but found that the reward function was too sensitive to the weights and setting the process of finding the weights was not as intuitive as our reward function. For the list of notations used to describe the MDP, see Table 2.

5. Reinforcement learning

Reinforcement learning finds the optimal policy π^* to be followed that maximizes the long term expected cumulated reward

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{i+k} \middle| s_i = s \right] \tag{2}$$

for the trajectory $(s_i, a_i, r_i, s_{i+1}, \ldots)$, $i \in \mathbb{N}$.

Here the notation $\mathbb{E}_\pi$ means that we compute the expectation considering an MDP under the policy π, and $V^\pi(s)$, often called the value

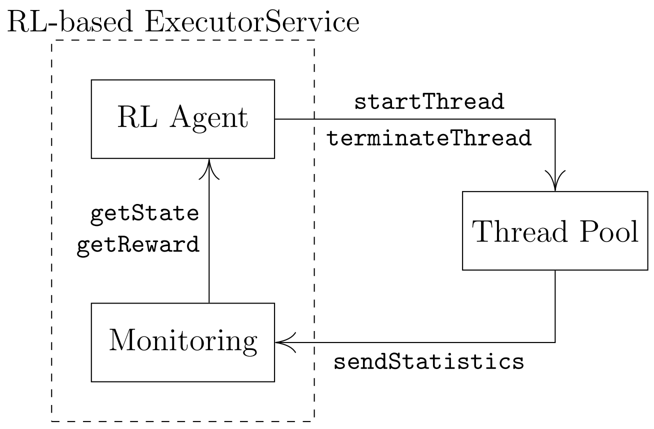

Fig. 3. Interaction between the RL agent and the thread pool.

function, is signifying the expectation of the total discounted reward if we follow policy π starting from state s. Note, that the optimal policy does not depend on the starting state.

Fig. 3 shows us a model of a RL agent integrated into an `ExecutorService` that controls a thread pool. In this setup, threads in the thread pool would send statistics on message delay, thread state, traffic load etc. A monitoring service would interpret these observations and calculate the current reward and state. The RL agent could decide to start or terminate threads in the thread pool based on its policy from the state information. Using the reward, the agent could improve its policy with the learning algorithm.

In this work, we used the *proximal policy optimization* (PPO) (Schulman et al., 2017) method to find the optimal policy. We also experimented with other methods like the *actor-critic* algorithm, or *deep q-learning* (Sutton and Barto, 2018) and found them often not as accurate or stable as the PPO algorithm.

5.1. A general learning framework

Algorithm 1 shows the learning framework we used for simulations and training. It displays the interaction between the learning agent and the simulation environment. The training runs for `tr_st` amount of steps, where in each step the agent executes an action and then stores the state, the action, the probability of the actions, the reward, and the next state to improve its policy.

Algorithm 1 General training loop

```
 1: Initialize system, and get initial state $s_0$.
 2: Initialize learning parameters of Agent.
 3: for tr_st steps do
 4:     Get action from agent: $a_i, \leftarrow \pi(s_i, \boldsymbol{\theta})$
 5:     Execute action $a_i$ to adjust thread pool size.
 6:     Observe the new state $s_{i+1}$ and performance measures after $\Delta T$ time.
 7:     Compute reward $r_i$ from the measurements using (1).
 8:     Agent.Store($s_i$, $a_i$, $\pi(\cdot|s_i, \boldsymbol{\theta})$, $r_i$, $s_{i+1}$)
 9:     Agent.Update( )
10:     $s_i \leftarrow s_{i+1}$, $i \leftarrow i + 1$
```

Here $\pi(s_i, \boldsymbol{\theta})$ is the policy function parameterized with vector $\boldsymbol{\theta}$. It maps the state s_i to a probability distribution over the possible actions, thus outputs a vector of size $|\mathcal{A}|$. We denote this vector with $\pi(\cdot|s_i, \boldsymbol{\theta})$. We get action a_i by sampling an action from $\mathcal{A}$ using this distribution given by the policy.

5.2. The proximal policy optimization algorithm

In the PPO algorithm, the RL agent uses the $\pi(s, \boldsymbol{\theta})$ policy to interact with the environment and adjusts the $\boldsymbol{\theta}$ parameter vector during training to find the optimal policy. Here $\pi(s, \boldsymbol{\theta})$ is implemented with a non-linear approximator, a neural network. Aside from the policy $\pi(s, \boldsymbol{\theta})$, the algorithm also keeps track of the value function by estimating it with $V(s, \boldsymbol{\omega})$, a function approximated with a second neural network with $\boldsymbol{\omega}$ parameter vector. Both the policy function's and the value function's neural network receive the state as an input, but the former outputs a vector of size $|\mathcal{A}|$, whereas the latter results in a single real value.

Algorithm 2 is similar to the one presented by Schulman et al. (2017). The algorithm keeps track of $\tilde{r}$, an estimate of the mean reward using soft updates. In each simulation step it runs k number of update steps and it stores a history of size b before running the update. Also, generalized advantage estimation (GAE) (Schulman et al., 2016) was used for estimating the advantage $\hat{A}^{\text{GAE}}$ and the reward scheme was modified for the average reward scenario for continuing tasks. We also used $H(\pi(\cdot|\mathbf{s}, \boldsymbol{\theta})) = -\sum_{a' \in \mathcal{A}} \pi(a'|\mathbf{s}, \boldsymbol{\theta}) \log \pi(a'|\mathbf{s}, \boldsymbol{\theta})$ as a regularization function. Note, that we used bold symbols to signify vector values and lower index j to represent their components (e.g.: $\mathbf{v} = [v_0, v_1, \ldots, v_{n-1}]$ for a vector $\mathbf{v}$ of size n). Here α_θ and α_ω are the learning rates of the neural networks approximating the policy and the value functions. ξ is the coefficient that scales the entropy. The hyperparameter α_R is the reward averaging factor, χ is the GAE's hyperparameter, which implicitly contains the discount factor γ, and ϵ is the clipping ratio used by PPO. The vector $\mathbf{r_t}(\theta)$ is the so-called probability ratio of the actions $\mathbf{a}$ and the symbol $\odot$ is the elementwise vector product. The notation $\pi(a|s, \boldsymbol{\theta})$ signifies the probability of action a in state s when the policy is parameterized with $\boldsymbol{\theta}$.

Algorithm 2 Proximal policy optimization update in the i-th decision epoch

```
 1: procedure Store($s_i$, $a_i$, $\pi(\cdot|s_i, \boldsymbol{\theta})$, $r_i$, $s_{i+1}$)
 2:     Append $(s_i, a_i, \pi(\cdot|s_i, \boldsymbol{\theta}), r_i, s_{i+1})$ to batch.
 3: procedure Update( )
 4:     if size of the batch $< b$ then return
 5:     Get batch $\mathbf{s}, \mathbf{a}, \boldsymbol{\pi}_{\text{old}}, \mathbf{r}, \mathbf{s}'$ of size $b$.
 6:     $\tilde{r} \leftarrow (1 - \alpha_R)\tilde{r} + \alpha_R \sum_{j=0}^{b-1} r_j / b$
 7:     for $k$ epochs do
 8:         for $j = 0, 1, \ldots, b$ do
 9:             $\text{TD}_{\text{target},j} \leftarrow r_j - \tilde{r} + V(s'_j, \boldsymbol{\omega})$
10:             $\delta_j \leftarrow \text{TD}_{\text{target},j} - V(s_j, \boldsymbol{\omega})$
11:             $\hat{A}_j^{\text{GAE}} \leftarrow \sum_{l=0}^{b-j-1} \chi^l \delta_{j+l}$
12:             $r_t(\boldsymbol{\theta})_j \leftarrow \pi(a_j|s_j, \boldsymbol{\theta}) / \pi_{\text{old},j}$
13:         $\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} + \alpha_\omega \mathbf{TD_{target}} \odot \nabla_\omega V(\mathbf{s}, \boldsymbol{\omega})$
14:         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta \nabla_\theta \min\{\mathbf{r_t}(\boldsymbol{\theta}) \odot \hat{\mathbf{A}}^{\mathbf{GAE}}, \text{clip}(\mathbf{r_t}(\boldsymbol{\theta}), 1-\varepsilon, 1+\varepsilon) \odot \hat{\mathbf{A}}^{\mathbf{GAE}}\} + \xi H(\pi(\cdot|\mathbf{s}, \boldsymbol{\theta}))$
15:     Clear batch storage.
```

5.3. Setting hyperparameters

In general, hyperparameter tuning is a costly operation in machine learning, and it is even more expensive in RL as it requires training data and comparisons in each tuning iteration. Two popular methods are grid search and random search (Bergstra and Bengio, 2012; Bergstra et al., 2011). In the former, hyperparameters have a set of fixed candidate values and training is run on every combination of them. In the latter, hyperparameters are given a range of values from which they are sampled randomly.

During the training, we found that for most hyperparameters, the literature's default values, recommended by Schulman et al. (2017), were sufficient for the learning process. We identified two hyperparameters to which the algorithm was sensitive. The κ reward coefficient and the ξ entropy weight. In the more simple experiments, we applied grid search for a few candidate values, but for the later experiments we used *population based training* (PBT) (Jaderberg et al., 2017).

PBT is a hyperparameter search method based on evolutionary algorithm principles. In short, it runs multiple (par) simulation instances in parallel for l steps, after which it executes so-called *exploitation* and *exploration* steps. Each simulation trains an RL agent, each with different sets of hyperparameters to tune. The algorithm evaluates each of the par agents during exploitation and sorts them according to a performance measure. A copy of the best agent replaces the worst agent. In the exploration phase, the hyperparameters of the new replacements are slightly perturbed. Finally, the training resumes for another l steps, and this is repeated m times.

This study used PBT to identify a small set of candidate κ and ξ values. After that, we ran a grid search on these values and picked the hyperparameter set with the best performance. Algorithm 3 contains the hyperparameter search we used with PBT. In line 7 we rank the agents by performance, which means that first, we sort them by whether they can keep the mean waiting time below δ_{th} and if they can, we sort them by increasing mean idle thread count, otherwise, we sort them by increasing mean waiting times. This results in $agent_0$ being the best performing agent, an agent that can keep the QoS level while having the lowest idle thread counts. In lines 9–11 we random sample c_κ and c_ξ variables from uniform distributions and use them to perturb the κ and ξ values of the new agent. Finally, we select the 3 and the 2 best performing κ and ξ values, respectively and run a grid search on every combination of them.

Algorithm 3 Hyperparameter search with population based training

```
1:  // Population Based Training
2:  Initialize list of environments $envs_i$, $i = 0, \dots, \mathrm{par}$.
3:  Initialize list of agents $agents_i$ with $\kappa_i$, $\xi_i$, $\theta_i$ and $\omega_i$, $i = 0, \dots, \mathrm{par}$.
4:  for $m$ iterations do
5:      Train $agents_i$ on $envs_i$ for $l$ steps using lines 3–10 of Algorithm 1. $i = 0, \dots, \mathrm{par}$.
6:      Evaluate agents.
7:      $agents \leftarrow \mathrm{sorted}(agents)$
8:      $\theta_{\mathrm{par}}, \omega_{\mathrm{par}}, \kappa_{\mathrm{par}}, \xi_{\mathrm{par}} \leftarrow \theta_0, \omega_0, \kappa_0, \xi_0$        ▷ Exploitation
9:      Sample $c_\kappa$ from $Uniform(1-\epsilon_\kappa, 1+\epsilon_\kappa)$.
10:     Sample $c_\xi$ from $Uniform(1-\epsilon_\xi, 1+\epsilon_\xi)$.
11:     $\kappa_{\mathrm{par}}, \xi_{\mathrm{par}} \leftarrow c_\kappa \kappa_{\mathrm{par}}, c_\xi \xi_{\mathrm{par}}$        ▷ Exploration
12: // Grid Search
13: Select $\kappa_0, \kappa_1, \kappa_2, \xi_0, \xi_1$ as candidates.
14: for $n$ iterations do
15:     Initialize list of environments $envs_{i,j}$, $i \in \{0,1,2\}$, $j \in \{0,1\}$.
16:     Initialize list of agents $agents_{i,j}$ with $\kappa_i$, $\xi_j$, $\theta_{i,j}$ and $\omega_{i,j}$, $i \in \{0,1,2\}$, $j \in \{0,1\}$.
17:     Train and evaluate $agents_{i,j}$ on $environment_{i,j}$, $i \in \{0,1,2\}$, $j \in \{0,1\}$.
18: Select $\kappa_i$ and $\xi_j$ for the best performing $agents_{i,j}$ on average.
```

Note that the PBT involves one continuous training process. We initialized the agents and the environments initially and did not reset the training process after the exploitation and exploration steps. However, during the grid search, training was performed for each combination of κ_i and ξ_j values for n times and the final hyperparameters were chosen based on the performance measures averaged from these n runs.

This search also introduces new hyperparameters, in this case par, l, m, ϵ_κ, ϵ_ξ, n. However, we found that setting these hyperparameters was much easier, and the training was less sensitive to their changes. For the complete list of notations used to describe RL, see Table 3.

6. Design of investigation and results

For our experiments, we created simulated environments to evaluate the timeout rule and the RL method. We want to answer the following questions: is there an advantage in using multiple actions; can the RL method compare to the timeout rule; and under what conditions does RL outperform the timeout rule.

6.1. Simulation setup

We assumed sessions arrive according to a Poisson process with arrival rate λ_{session}. This is a reasonable assumption when working with a high number of IoT devices (Metzger et al., 2019). We considered a case with fixed arrival rates and another case with arrival rate varying through time. In the case of fixed arrival we considered a low traffic scenario with $\lambda_{\text{session}} = 1\frac{1}{s}$ and a high traffic scenario with $\lambda_{\text{session}} = 10\frac{1}{s}$.

For the case of varying arrival rate we used

$$\lambda_{\text{session}}(t) = 16.5038 + 8.55524\sin\left(\frac{\pi}{12}t + 3.08\right) + 5.00952\sin\left(\frac{\pi}{6}t + 2.08\right) + 1.58857\sin\left(\frac{\pi}{4}t + 1.14\right). \quad (3)$$

Note that equation (3) is borrowed from Wang et al. (2015) and scaled to $\lambda_{\text{session}} \in [0, 30]$. The purpose of scaling is to have the range of arrival rates comparable to the fixed arrival rate experiments. It is worth mentioning that function (3) was created using mobile network data, however, we believe it represents traffic created by human activity well and therefore is also suitable for our IoT experiments. We also assumed that the length of the session is distributed exponentially with mean $\eta = 2$ s. The maximum number of sessions in the system was set to $L = 100$.

When a session arrives, an actor is spawned for it with its own message queue. During their lifetime, each session sends packets with inter-arrival time $\tau_{\text{arrive}} = 0.1$ s. The service time t_{service} of a packet follows

$$t_{\text{service}}(B(t)) = \begin{cases} \tau_{\text{service}} & \text{if } B(t) \le y_{\text{lim}} \\ \frac{B(t)}{y_{\text{lim}}}\tau_{\text{service}} & \text{if } B(t) > y_{\text{lim}}, \end{cases} \quad (4)$$

where $\tau_{\text{service}} = 0.05$ s is the base service time of each packet. We assumed the system has 32 CPU cores, that is, $y_{\text{lim}} = 32$. While the number of busy threads B is below y_{lim}, the service time is no different from the base value. However, once B exceeds the available number of CPU cores, a contention starts for CPU time slots resulting in lower service times. The processing of the packet starts immediately on arrival if the queue is empty. Otherwise, it is stashed in the queue in a FIFO manner. We used bounded message queues and limited the queue sizes to $M = 20$.

The worker threads do the processing of the packets in the thread pool. When a worker finishes its job, it looks for the next session waiting for a thread in a round-robin manner. The maximum size of the thread pool was set to $N = 64$. Furthermore, we also set a minimum thread pool size of 2 so that the RL method would not terminate all threads while exploring the state space. We assumed turning off a thread happens immediately. However, turning them on takes $t_{\text{start}} = 0.1$ s.

6.2. Comparison of the timeout rule and the RL method at fixed arrival rates

When running the timeout rule we considered three timeout cases: 1 s, 10 s, and 60 s. For both the fixed and the varying arrival rate case, we simulated a 24 hour period and collected the performance measures for this period.

When RL is applied, we investigated the *on/off* and the *multi-action* action selection alternatives. For the reward function, we set the blocking threshold to $p_{b,th} = 0.01$ and we considered two cases for the delay threshold δ_{th}: a case with a lower QoS requirement $\delta_{th} = 0.1$ s; and

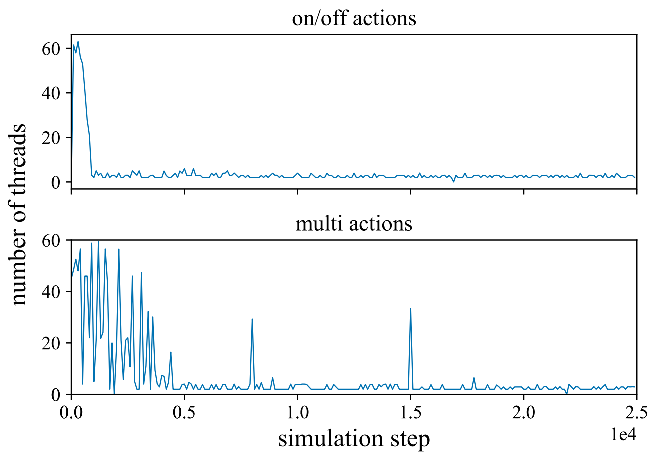

Fig. 4. Thread counts during the training of the AC algorithm for different environments at $\lambda_{\text{session}} = 1\frac{1}{s}$, $1/\mu_{\text{session}} = 2$ s, $\delta_{th} = 0.1$, and $\xi = 0$.

Table 4
Hyperparameters of PPO runs.

Name	Value
Neural network hidden layers	1
Hidden node count	50
Value function learning rate (α_ω)	0.001
Policy function learning rate (α_θ)	0.001
Reward averaging (α_R)	0.1
Batch size (b)	32
Epoch count (k)	5
Clipping parameter (ϵ)	0.1
GAE parameter (χ)	0.9
PBT parallel instances (par)	8
PBT iterations (m)	50
PBT training steps (l)	43200
PBT κ perturbation range (ϵ_κ)	0.35
PBT ξ perturbation range (ϵ_ξ)	0.25
Grid search run count (n)	10

Table 5
Comparison of the timeout rule and the PPO with $\xi = 0$ at $\lambda_{\text{session}} = 1\frac{1}{s}$.

Timeout rule			
Timeout	Mean thread count	Mean idle threads	Mean waiting time
1 s	2.501584	1.520005	0.003406
10 s	3.906392	2.951080	0.000626
60 s	5.276596	4.330228	0.000087

RL (PPO, $\xi = 0$)				
δ_{th} [s]	Action	Mean thread count	Mean idle threads	Mean waiting time
0.1	on/off	2.126357	1.126818	0.013988
	multi	2.388725	1.392581	0.012177
0.01	on/off	3.665027	2.672725	0.000503
	multi	3.779104	2.788579	0.000412

Table 6
Comparison of the timeout rule and the PPO with $\xi = 0$ at $\lambda_{\text{session}} = 10\frac{1}{s}$.

Timeout rule			
Timeout	Mean thread count	Mean idle threads	Mean waiting time
1 s	15.433190	5.664380	0.000854
10 s	19.102335	9.360207	0.000136
60 s	22.163054	12.402659	0.000019

RL (PPO, $\xi = 0$)				
δ_{th} [s]	Action	Mean thread count	Mean idle threads	Mean waiting time
0.1	on/off	11.595852	1.853114	0.056434
	multi	10.933337	1.190067	0.058022
0.01	on/off	16.200938	6.449216	0.000554
	multi	14.945809	5.186084	0.000618

a case with a higher QoS requirement $\delta_{th} = 0.01$ s. In case of $\delta_{th} = 0.1$ s we set $\kappa = 1$ and in case of $\delta_{th} = 0.01$ s we set $\kappa = 100$.

We used a neural network (NN) to approximate the policy and the value function. The NN has an input layer accepting state vectors. This is fully connected with the hidden layer, where the activation function is a rectified linear unit (ReLU). These two layers are shared between the policy and the value functions. For the former, the output is a layer fully connected to the hidden layer with nodes for each action. The result is converted to probabilities with a sigmoid activation function. A layer with a single node is fully connected to the hidden layer for the latter, and no activation function is used. Values for the training hyperparameters can be found in Table 4.

We ran training for 500 000 time steps and then evaluated the resulting RL agent for 100 000 time steps with $\Delta T = 1$ s. Fig. 4 shows us the number of threads during training for the first 25 000 steps for the on/off and the multi-action schemes. We can see that the RL agent could converge to the optimal policy under both action selection schemes. With the on/off actions, it took around 1000 steps, whereas with multi actions it took 5000 steps to converge because the multi-action action selection involves a much larger action space than the on/off action scheme.

Tables 5 and 6 show the results of the simulation runs under the timeout scheme and under the PPO agent. We displayed the mean number of total threads and idle threads for each run and the mean waiting time of the packets. We chose to omit the blocking rate. Blocking seldom happened due to the message queue being large enough most of the time.

Looking at the results of the timeout rule, we can see the trade-off between the number of threads and the mean waiting time. We can see that the RL method could minimize the idle thread count while keeping the waiting times below δ_{th}.

6.3. Comparison of the timeout rule and the RL method at varying arrival rates

For the case of varying arrival rates we used a simple sine function $\lambda_{\text{session}}(t) = 12.5 + 12.5\sin\left(\frac{\pi}{6}t\right)$, and ran 1209 600 of training steps with a time step of $\Delta T = 1$ s which is 2 weeks in simulation time. Then we used (3) to evaluate the RL agent for 86400 time steps which is 1 day of simulation time. Fig. 5 shows us the graphs for the $\lambda_{\text{session}}(t)$ used during training and evaluating.

During training we used entropy regularization and did a grid search on ξ for $\xi \in \{0, 0.01, 0.1\}$. We also ran each training scenario 10 times and averaged the performance measures of the evaluated RL agent. It is important to note, that occasionally the RL method got stuck and produced erroneous values that we needed to exclude. These cases were easy to identify as the resulting performance measures were unrealistically different from the rest. We picked $\xi = 0.01$, which produced the best results.

In Table 7 we can see the results with the varying arrival rates. In the case of the RL experiments, we can see that using the multi-action selection method performed worse than the on/off actions and in the case of $\delta_{th} = 0.01$ s the agent could not keep the main waiting time below the threshold in some of the runs. This is because the change in λ_{session} through time was slow enough compared to ΔT that turning on and off single threads were enough to react. Another reason is that the probability of the actions cannot go down to zero. This is caused by the entropy regularization, which prevents degenerate distributions for the action selection. As a consequence, the mistake the agent can do with the multi-action space can be much bigger. We can also see this in Fig. 6 which shows that controlling the number of threads was easier with the on/off actions. By restricting the action space, we could

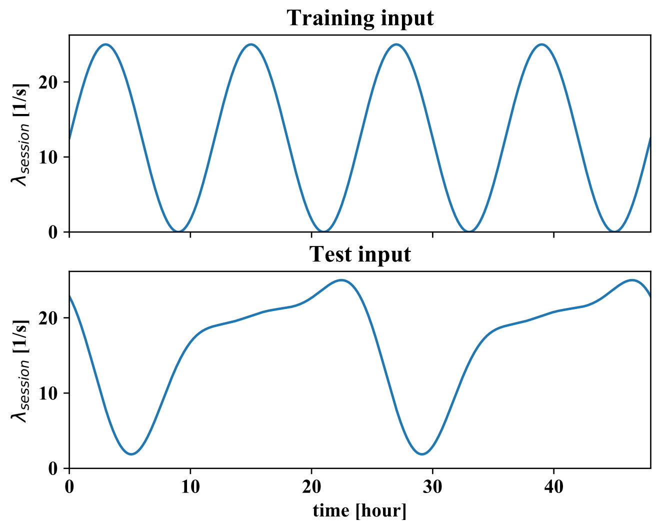

Fig. 5. Arrival rate used for simulation. Top used for training, bottom used for evaluation of RL and timeout rules.

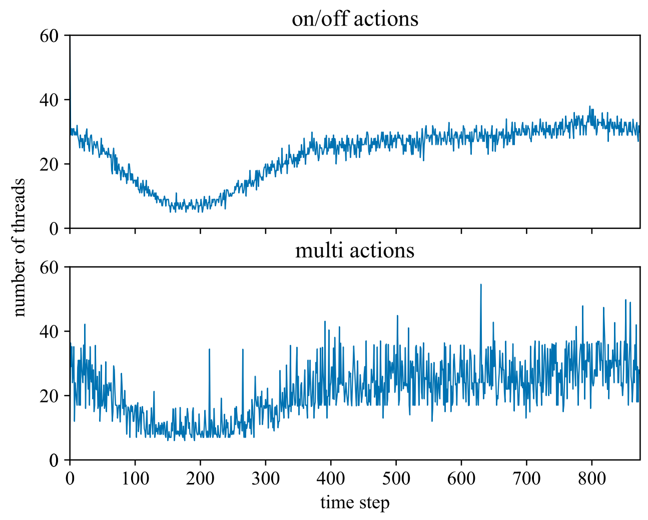

Fig. 6. Number of threads during the evaluation phase of the RL agent with $\xi = 0.01$ and $\delta_{th} = 0.1$.

Table 7
Comparison of the timeout rule and the PPO with $\xi = 0.01$ with varying λ_{session}.

Timeout rule			
Timeout	Mean thread count	Mean idle threads	Mean waiting time
1 s	23.275186	7.099843	0.000773
10 s	27.675422	11.723045	0.000221
60 s	32.305623	16.138733	0.000108

RL (PPO $\xi = 0.01$)				
δ_{th} [s]	Action	Mean thread count	Mean idle threads	Mean waiting time
0.1	on/off	18.200269	2.108050	0.062803
	multi	22.775796	6.554406	0.037386
0.01	on/off	23.408005	7.277181	0.002342
	multi	22.819045	6.759416	0.065778

reduce the error the PPO agent made. Also, the results provided by the RL method are just slightly worse if we compare the results for the 1 s timeout with the 0.01 s threshold, where the timeout rule gave fewer idle threads and slightly lower waiting times. However, if we did not need such strict waiting times but wanted to further decrease resource usage, we could do it easily with the RL method. In case of the timeout rule, we would need to search again for a timeout value in a lower region. As we can see in Table 7, by setting a 0.1 s threshold we could much lower the mean thread count.

Table 8
Results for the timeout scheme under different t_{start} values with varying λ_{session}.

t_{start}	Timeout	Mean thread count	Mean idle threads	Mean waiting time
1 s	1 s	23.183291	7.004011	0.038398
	10 s	28.113262	11.971182	0.006512
	60 s	32.163872	16.023724	0.001145
5 s	1 s	22.476157	6.929937	0.326515
	10 s	28.394714	12.386796	0.056084
	60 s	32.191699	16.075730	0.007978
10 s	1 s	22.336707	6.743414	0.661729
	10 s	28.560191	12.544235	0.124329
	60 s	32.270030	16.157706	0.017190

6.4. Performance under different t_{start} values

Looking at the results of the timeout scheme in Tables 5–7 we see that the lower the timeout value, the lower the idle thread count, whereas the mean waiting time is still below the threshold $\delta_{th} = 0.1$ s. One might ask, why not further decrease the timeout value? In this particular case, it would be possible to decrease the timeout to get better results. However, after a point, the improvement would stop. If the timeout was very low, threads would be stopped shortly after finishing a job. This means that new incoming packets would arrive into a system without idle threads, and they would need to wait for a new thread to start to get processed. Thus, waiting time would depend on t_{start}, the time it takes a thread to start. In Table 8 we can see the results of our experiments with the timeout scheme with different t_{start} values. It is obvious that decreasing the timeout value would increase the waiting time. However, at higher t_{start} values the mean waiting time reaches the threshold much sooner. For example if we set $\delta_{th} = 0.1$ s, at $t_{\text{start}} = 5$ s the ideal timeout, where the mean waiting time is just below the threshold level, would be between 1 and 10 seconds according to Table 8. At $t_{\text{start}} = 10$ s this value would be between 10 and 60 seconds. Thus, we would need an algorithm, like a binary search, to find an ideal timeout value for our system.

For comparison, we trained the PPO agent on the environment under different t_{start} values, Table 9 shows the results for on/off actions and Table 10 shows them for multi-actions. For these runs, we used Algorithm 3 to find the ξ and κ hyperparameters. Looking at the results of the on/off actions, we can see that the agent could keep the mean waiting time below the threshold level while minimizing the number of idle threads. The RL agent could still maintain the mean waiting time below the threshold level with multi-actions and still performed better than the timeout method. However, it could not minimize the number of idle threads as well as with the on/off actions due to the much bigger action space the agent needs to explore and the entropy regularization that prevents zero probability actions. Another thing we can notice is that for higher t_{start} values, the timeout scheme needs a lot more threads to meet the QoS requirement than the RL agent.

Fig. 7 shows that the RL method performs much better during the evaluation phase. A few idle threads can still maintain waiting times below the threshold. Note that the time between two decisions is $\Delta T = 1$ s. If t_{start} is larger than $\Delta T = 1$ s, a thread will not be ready at the next decision timestep and the recent decision will not have immediate impacts. Nevertheless, RL can still deal with these delayed rewards because the value function of state s_i, denoted by $V^{\pi}(s_i)$, contains the next reward r_i and recursively contains the value function of the next state $V^{\pi}(s_{i+1})$. This endless recursion guarantees that if effects of a_i only appear in a later reward r_{i+k}, it will still be included in the value of s_i. We can observe that with the timeout method, higher timeout values can guarantee low waiting times; however, they result in higher idle

Table 9
Results for PPO agent with on/off actions under different t_{start} values with varying λ_{session}, $\delta_{th} = 0.01$ s.

t_{start}	ξ	κ	Mean thread count	mean idle threads	Mean waiting time
1 s	0.0033	25.4578	21.9863	5.8618	0.003700
5 s	0.0137	19.2551	22.8223	6.7079	0.006921
10 s	0.1000	10.0000	23.9722	7.8588	0.004386

Table 10
Results for PPO agent with multiple actions under different t_{start} values with varying λ_{session}, $\delta_{th} = 0.01$ s.

t_{start}	ξ	κ	Mean thread count	mean idle threads	Mean waiting time
1 s	0.0752	15.9797	23.4559	7.3053	0.006267
5 s	0.0116	4.6571	28.4877	12.3906	0.007994
10 s	0.0104	9.3111	42.4358	26.2788	0.003966

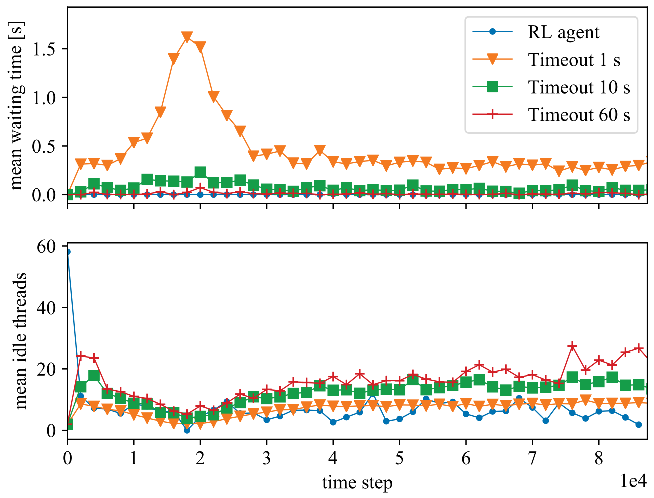

Fig. 7. Comparison of the mean waiting time and the mean idle thread count during the evaluation phase at $t_{\text{start}} = 5$ s and $\delta_{th} = 0.01$.

thread counts. The reason is that the policy of the RL decides based on mean traffic measurements, and it learned through averaging multiple rewards in a given state.

7. Conclusion

We have investigated the dynamic scaling of the size of a thread pool for an actor-based IoT application. We have formulated the problem as an MDP by defining its state space, action space, and reward function. We have used the PPO algorithm to find the optimal policy and used PBT to find the proper hyperparameter set under various scenarios. We have found that broadening the action space by allowing the agent to turn multiple threads on or off at the same time did not improve results. Results show that the timeout scheme falls off when threads take longer, whereas the RL approach still holds in these scenarios.

CRediT authorship contribution statement

Hai T. Nguyen: Methodology, Software, Experiment, Investigation, Evaluation of results, Writing. **Tien V. Do:** Conceptualization, Methodology, Evaluation of results, Writing, Reviewing and editing, Supervision. **Csaba Rotter:** Conceptualization, Writing, Review and Editing, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The work of H. Nguyen is partially supported by the National Research Development and Innovation Office of Hungary through the OTKA K-123914 project. The research reported in this paper and carried out at the Budapest University of Technology and Economics has been partially supported by the National Research Development and Innovation Fund based on the charter of bolster issued by the National Research Development and Innovation Office under the auspices of the Ministry for Innovation and Technology.

We thank Editor and anonymous reviewers whose comments were helpful to improve the presentation of the paper.

References

Aazam, M., Huh, E., 2015. Fog computing micro datacenter based dynamic resource estimation and pricing model for IoT. In: 2015 IEEE 29th International Conference on Advanced Information Networking and Applications, pp. 687–694.

Akka documentation, version 2.6.13, (Accessed: 29 March 2021) [Online]. Available: https://doc.akka.io/docs/akka/current/typed/guide/introduction.html.

Alazab, M., Khan, S., Krishnan, S.S.R., Pham, Q., Reddy, M.P.K., Gadekallu, T.R., 2020. A multidirectional LSTM model for predicting the stability of a smart grid. IEEE Access 8 (85), 454–485, 463.

Alazab, M., Lakshmanna, K., T. R. G, Pham, Q.-V., Reddy Maddikunta, P.K., 2021. Multi-objective cluster head selection using fitness averaged rider optimization algorithm for iot networks in smart cities. Sustain. Energy Technol. Assess. 43, 100973, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2213138820314016.

Albahri, A., Alwan, J.K., Taha, Z.K., Ismail, S.F., Hamid, R.A., Zaidan, A., Albahri, O., Zaidan, B., Alamoodi, A., Alsalem, M., 2021. IoT-based telemedicine for disease prevention and health promotion: State-of-the-art. J. Netw. Comput. Appl. 173, 102873, [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804520303374.

Arteaga, C.H.T., Rissoi, F., Rendon, O.M.C., 2017. An adaptive scaling mechanism for managing performance variations in network functions virtualization: A case study in an NFV-based EPC. In: 2017 13th International Conference on Network and Service Management (CNSM), pp. 1–7.

Bandyopadhyay, D., Sen, J., 2011. Internet of things: Applications and challenges in technology and standardization. Wirel. Pers. Commun. 58 (1), 49–69, [Online]. Available: https://doi.org/10.1007/s11277-011-0288-5.

Barcelo, M., Correa, A., Llorca, J., Tulino, A.M., Vicario, J.L., Morell, A., 2016. Iot-cloud service optimization in next generation smart environments. IEEE J. Sel. Areas Commun. 34 (12), 4077–4090.

Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B., 2011. Algorithms for hyper-parameter optimization. In: Proceedings of the 24th International Conference on Neural Information Processing Systems. In: ser. NIPS'11, Curran Associates Inc., Red Hook, NY, USA, pp. 2546–2554.

Bergstra, J., Bengio, Y., 2012. Random search for hyper-parameter optimization. J. Mach. Learn. Res. 13, 281–305.

Bibal Benifa, J.V., Dejey, D., 2019. Rlpas: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment. Mob. Netw. Appl. 24, 1348–1363.

Bonér, J., 2010. Introducing Akka, [Online]. Available: http://jonasboner.com/introducing-akka/ (Accessed: 29 March 2021).

Chen, N., Lin, P., 2010. A dynamic adjustment mechanism with heuristic for thread pool in middleware. In: 2010 Third International Joint Conference on Computational Science and Optimization, vol. 1, pp. 369–372.

Cheng, M., Li, J., Nazarian, S., 2018. Drl-cloud: Deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers. In: 2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 129–134.

Chowdhury, A., Raut, S.A., Narman, H.S., 2019. Da-drls: Drift adaptive deep reinforcement learning based scheduling for iot resource management. J. Netw. Comput. Appl. 138, 51–65, [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804519301341.

Diaz Sánchez, D., Simon Sherratt, R., Arias, P., Almenarez, F., A., Marín, 2015. Enabling actor model for crowd sensing and IoT. In: 2015 International Symposium on Consumer Electronics (ISCE), pp. 1–2.

García-Valls, M., 2016. A proposal for cost-effective server usage in cps in the presence of dynamic client requests. In: 2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC), pp. 19–26.

Gartner, Press release, https://www.gartner.com/newsroom/id/3598917, (Accessed: 05 September 2019).

Haubenwaller, A.M., Vandikas, K., 2015. Computations on the edge in the internet of things. Procedia Comput. Sci. 52, 29–34, the 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S187705091500811X.

Hewitt, C., 2010. Actor model of computation: Scalable robust information systems. arXiv e-prints, arXiv:1008.1459.

Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W.M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., Fernando, C., Kavukcuoglu, K., 2017. Population based training of neural networks.

Java TM Platform, standard ed. 7, (Accessed: 29 March 2021) [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html.

Jin, Y., Bouzid, M., Kostadinov, D., Aghasaryan, A., 2018. Model-free resource management of cloud-based applications using reinforcement learning. In: 2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), pp. 1–6.

Jin, Y., Kostadinov, D., Bouzid, M., Aghasaryan, A., 2019. Common structures in resource management as driver for reinforcement learning: A survey and research tracks. In: Renault, É., Boumerdassi, P. Mühlethalerand S. (Eds.), Machine Learning for Networking. Springer International Publishing, Cham, pp. 117–132.

Kang, D., Han, S., Yoo, S., Park, S., 2008. Prediction-based dynamic thread pool scheme for efficient resource usage. In: 2008 IEEE 8th International Conference on Computer and Information Technology Workshops. pp. 159–164.

Kleinrock, L., 2018. Internet congestion control using the power metric: Keep the pipe just full, but no fuller. Ad Hoc Netw. 80, 142–157, [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1570870518302476.

Lee, K., Pham, H.N., Kim, H., Youn, H.Y., Song, O., 2011. A novel predictive and self – adaptive dynamic thread pool management. In: 2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications, pp. 93–98.

Levine, S., Pastor, P., Krizhevsky, A., Ibarz, J., Quillen, D., 2018. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. Int. J. Robot. Res. 37 (45), 421–436, [Online]. Available: https://doi.org/10.1177/0278364917710318.

Liu, C., Xu, X., Hu, D., 2015. Multiobjective reinforcement learning: A comprehensive overview. IEEE Trans. Syst. Man Cybern.: Syst. 45 (3), 385–398.

Mao, H., Alizadeh, M., Menache, I., Kandula, S., 2016. Resource management with deep reinforcement learning. In: Proceedings of the 15th ACM Workshop on Hot Topics in Networks. In: ser. HotNets '16, ACM, New York, NY, USA, pp. 50–56, [Online]. Available: http://doi.acm.org/10.1145/3005745.3005750.

Mao, H., Negi, P., Narayan, A., Wang, H., Yang, J., Wang, H., Marcus, R., r. Addanki, Khani Shirkoohi, M., He, S., Nathan, V., Cangialosi, F., Venkatakrishnan, S., Weng, W.-H., Han, S., Kraska, T., Alizadeh, M., 2019a. Park: An open platform for learning-augmented computer systems. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché Buc, F., Fox, E., Garnett, R. (Eds.), Advances in Neural Information Processing Systems, vol. 32. Curran Associates, Inc., pp. 2494–2506, [Online]. Available: http://papers.nips.cc/paper/8519-park-an-open-platform-for-learning-augmented-computer-systems.pdf.

Mao, H., Venkatakrishnan, S.B., Schwarzkopf, M., Alizadeh, M., 2019b. Variance reduction for reinforcement learning in input-driven environments. In: International Conference on Learning Representations. [Online]. Available: https://openreview.net/forum?id=Hyg1G2AqtQ.

Metzger, F., Hoßfeld, T., Bauer, A., Kounev, S., Heegaard, P.E., 2019. Modeling of aggregated IoT traffic and its application to an IoT cloud. Proc. IEEE 107 (4), 679–694.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D., 2015. Human-level control through deep reinforcement learning. Nature 518, 529 EP –.

Musaddiq, A., Zikria, Y.B., Hahm, O., Yu, H., Bashir, A.K., Kim, S.W., 2018. A survey on resource management in iot operating systems. IEEE Access 6, 8459–8482.

Ogasawara, T., 2008. Dynamic thread count adaptation for multiple services in smp environments. In: 2008 IEEE International Conference on Web Services, pp. 585–592.

Roestenburg, R., Bakker, R., Williams, R., 2015. Akka in Action, first ed. Manning Publications Co., Greenwich, CT, USA.

Schulman, J., Moritz, P., Levine, S., Jordan, M.I., Abbeel, P., 2016. High-dimensional continuous control using generalized advantage estimation. In: Bengio, Y., LeCun, Y. (Eds.), 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May (2016) 2-4, Conference Track Proceedings. [Online]. Available:.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O., 2017. Proximal policy optimization algorithms. CoRR, arXiv:abs/1707.06347 [Online]. Available: http://arxiv.org/abs/1707.06347.

Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D., 2016. Mastering the game of go with deep neural networks and tree search. Nature 529, 484 EP –.

Singh, P., Gupta, P., Jyoti, K., Nayyar, A., 2019. Research on auto-scaling of web applications in cloud: survey, trends and future directions. Scal. Comput.: Pract. Exp. 20 (2), 399–432.

Skarlat, O., Schulte, S., Borkowski, M., Leitner, P., 2016. Resource provisioning for IoT services in the fog. In: 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), pp. 32–39.

Sutton, R.S., Barto, A.G., 2018. Reinforcement Learning: An Introduction, second ed. The MIT Press, [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html.

Sutton, R.S., McAllester, D., Singh, S., Mansour, Y., 2000. Policy gradient methods for reinforcement learning with function approximation. In: Advances in Neural Information Processing Systems.

Tahsien, S.M., Karimipour, H., Spachos, P., 2020. Machine learning based solutions for security of internet of things (IoT): A survey. J. Netw. Comput. Appl. 161, 102630, [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804520301041.

Tesauro, G., Das, R., Chan, H., Kephart, J., Levine, D., Rawson, F., Lefurgy, C., 2008. Managing power consumption and performance of computing systems using reinforcement learning. In: Platt, J.C., Koller, D., Singer, Y., Roweis, S.T. (Eds.), Advances in Neural Information Processing Systems 20. Curran Associates, Inc., pp. 1497–1504, [Online]. Available: http://papers.nips.cc/paper/3251-managing-power-consumption-and-performance-of-computing-systems-using-reinforcement-learning.pdf.

ThingsBoard, ThingsBoard, Inc., https://thingsboard.io/ [Online] Available: https://thingsboard.io/.

Wang, S., Zhang, X., Zhang, J., Feng, J., Wang, W., Xin, K., 2015. An approach for spatial–temporal traffic modeling in mobile cellular networks. In: 2015 27th International Teletraffic Congress, pp. 203–209.

Watkins, C.J.C.H., Dayan, P., 1992. Q-learning. Mach. Learn. 8 (3), 279–292, [Online]. Available: https://doi.org/10.1007/BF00992698.

Wei, Y., Kudenko, D., Liu, S., Pan, L., Wu, L., Meng, X., 2019. A reinforcement learning based auto-scaling approach for saas providers in dynamic cloud environment. Math. Probl. Eng..

Xu, L.D., He, W., Li, S., 2014. Internet of things in industries: A survey. IEEE Trans. Ind. Inf. 10 (4), 2233–2243.

Zhang, H., Liu, N., Chu, X., Long, K., Aghvami, A., Leung, V.C.M., 2017. Network slicing based 5G and future mobile networks: Mobility, resource management, and challenges. IEEE Commun. Mag. 55 (8), 138–145.

Hai T. Nguyen received B.Sc. in Electrical Engineering and M.Sc. in Electrical Engineering from Budapest University of Technology and Economics, Budapest, Hungary in 2014 and 2016 respectively. His research interests are machine learning and computer networks.

Tien Van Do received the M.Sc. and Ph.D. degrees in telecommunications engineering from the Technical University of Budapest, Hungary, in 1991 and 1996, respectively. He is a professor in the department of Networked Systems and Services, the Budapest University of Technology and Economics. He habilitated at the Budapest University of Technology and Economics and received the DSc title from the Hungarian Academy of Sciences in 2011.

He led various projects on network planning and software implementations that results are directly used for industry such ATM & IP network planning software for Hungarian Telekom, GGSN tester for Nokia, performance testing program for the performance testing of the NOKIA's IMS product, automatic software testing framework for Nokia Siemens Networks. His research interests are queuing theory, telecommunication networks, cloud computing, performance evaluation and planning of ICT Systems and machine learning.

Csaba Rotter is heading 'Multi Cloud Orchestration' research group in Nokia Bell Labs. He is involved in different cloud related research topics targeting cloud native network service operation challenges in highly distributed multi-vendor environment. His special interest is on how to ensure performance related service level agreements for concurrent applications sharing the resources in the same distributed environment. His passion in automation started years before, when he was responsible for test automation concept development in large telecommunication systems. He received M.Sc. in applied electronics at the Technical University of Oradea in 1995 and M.Sc. in IT management at the Central European University Budapest in 2008. He joined Nokia in 1999, later Nokia Research Center (currently Nokia Bell Labs) in 2008.