Demonstrating a System for Dynamically Meeting Management Objectives on a Service Mesh

Forough Shahab Samani†, Kim Hammar†, and Rolf Stadler†
† Dept. of Computer Science, KTH Royal Institute of Technology, Sweden
Email: {foro, kimham, stadler}@kth.se

***Abstract*—We demonstrate a management system that lets a service provider achieve end-to-end management objectives under varying load for applications on a service mesh based on the Istio and Kubernetes platforms. The management objectives for the demonstration include end-to-end delay bounds on service requests, throughput objectives, and service differentiation. Our method for finding effective control policies includes a simulator and a control module. The simulator is instantiated with traces from a testbed, and the control module trains a reinforcement learning (RL) agent to efficiently learn effective control policies on the simulator. The learned policies are then transfered to the testbed to perform dynamic control actions based on monitored system metrics. We show that the learned policies dynamically meet management objectives on the testbed and can be changed on the fly.**

***Index Terms*—Performance management, reinforcement learning (RL), service mesh, digital twin, Istio, Kubernetes**

I. Introduction

End-to-end performance objectives for a service are difficult to achieve on a shared and virtualized infrastructure. This is because the service load often changes in an operational environment, and service platforms do not offer strict resource isolation, so that the resource consumption of various tasks running on a platform influences the service quality.

To continuously meet performance objectives for a service, such as bounds on delays or throughput for service requests, the management system must dynamically perform control actions that re-allocate the resources of the infrastructure. Such control actions can be taken on the physical, virtualization, or service layer, and they include horizontal and vertical scaling of computing resources, function placement, as well as request routing and request dropping.

Current solutions for resource allocation and configuration are either static configurations [1] or threshold-based dynamic solutions [2], both of which have limitations. The static solutions are typically tailored for each service running on a system and tend to over-provision the available resources, which can cost service providers excessively. Similarly, the dynamic solutions are limited to case-specific problems (e.g. scaling of CPU allocation) and depend on domain expertise.

As we demonstrate in this paper, a promising approach to address the above limitations is to learn effective control policies through Reinforcement Learning (RL). Specifically, in this demo paper, we describe a management system based on the RL framework presented in [3]. In our system, an RL agent continuously monitors the target system and takes control actions dynamically to meet management objectives. We show that the system is able to continuously adapt to changing objectives on a testbed that implements a service mesh based on Kubernetes [4] and Istio [5] (see Fig. 1). We also open-source the code [6], explain the testbed, and describe our RL method.

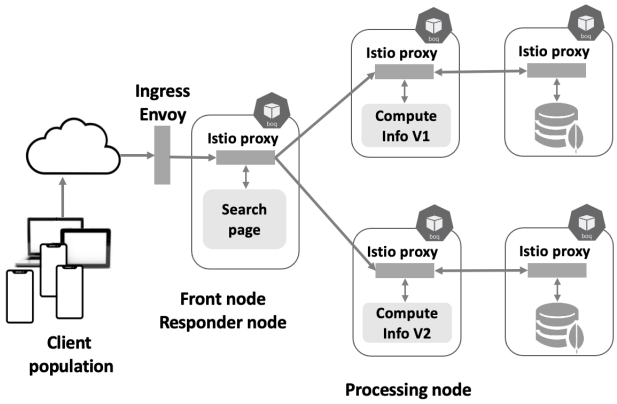

Fig. 1. Architecture of the microservice-based system deployed on the service mesh, which we use to demonstrate the management system.

II. Demonstration

We demonstrate the framework on a service mesh that includes application services that execute concurrently. We consider application services built from microservice components, whereby each component performs a unique function. A service request traverses a path on a directed graph whose nodes offer microservices. A service is realized as a contiguous subgraph on the service mesh. While the framework applies to the general service mesh, we perform the demonstration on the smaller configuration shown in Figure 2, right window.

We use our management system to demonstrate the following:

1) Evolution of system metrics (e.g. end-to-end delay and throughput) for a given management objective.
2) Reaction of the system to changing load conditions, e.g. reaction to a change from a smooth periodic load pattern to a probabilistic pattern with sudden changes.
3) Reaction of the system to changes in management objectives e.g. the reaction to a change from a management objective that maximizes overall throughput to a man-

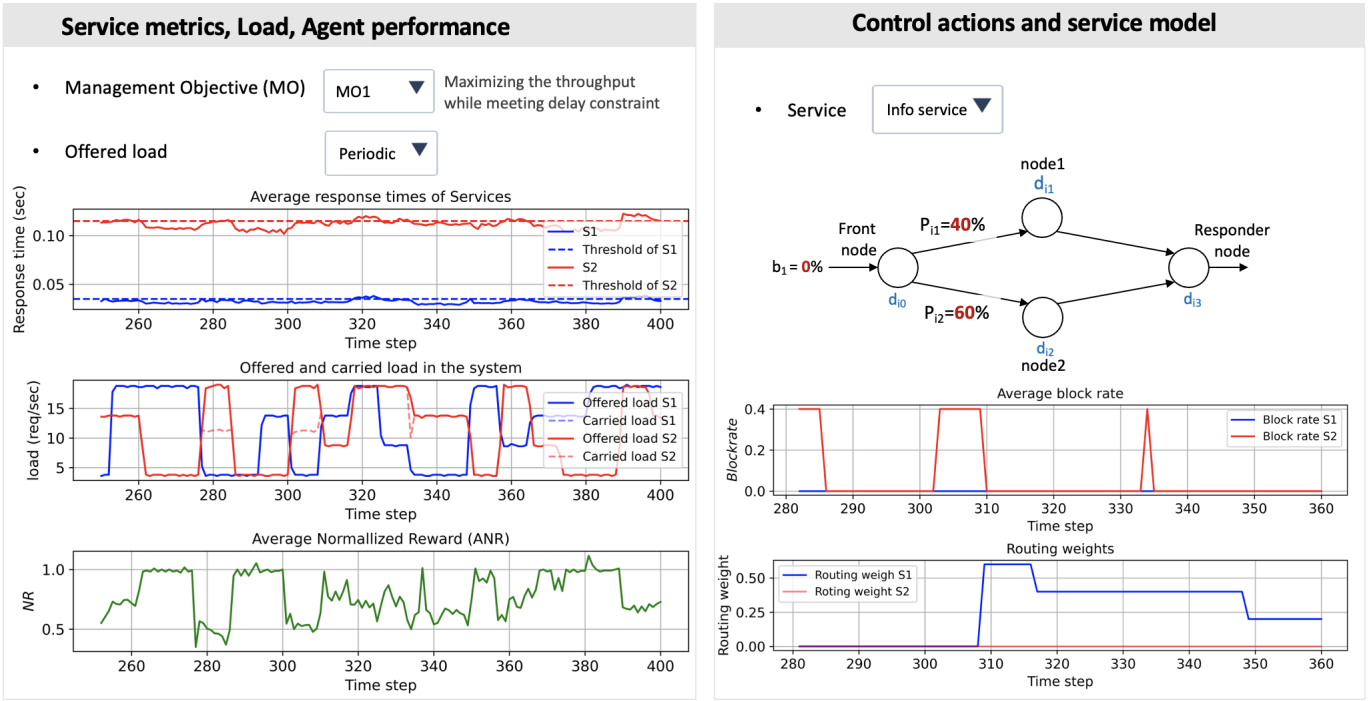

Fig. 2. Demonstration window: on the left side the service metrics, such as offered and carried load, response times of services, and performance of the agent for the given management objective are presented; on the right side, at the top, the control parameters for the selected service are presented, and at the bottom, time series of the actions taken by the agent is shown.

agement objective that prioritizes one service versus the other.

To demonstrate all features listed above in a short amount of time, we use prerecorded traces from the scenarios described in [3].

Figure 2 pictures the main user interface of the management system. The left window shows the system state. The state consists of various service metrics, e.g. response times and loads, which evolve in discrete time-steps. The left window also shows the performance of the control policy in terms of the reward function, which encodes the management objective. At the top of the left window, a user can change the load pattern and the management objective on the fly, which causes the learned control policy to automatically adjust to the changes.

The right window shows the service mesh configuration, which is visualized as a directed graph with the routing weights shown next to the edges, as well as the time series of actions taken by the control policy.

The demonstration provides insight into the learned control policies. We observe how control actions affect the system's performance and how they relate to the management objectives. We also examine how the control policies react to dynamic changes in the system, e.g. changes in the management objective and changes in the load pattern.

For example, we can observe that when the management objective (MO) is to maximize the overall load while keeping response times below a certain threshold (see MO1 in Section III), the control policy prioritizes services with a short response time, whereas when the management objective is to maximize service utilities (see MO2 in Section III), the policy prioritizes services with a high utility (e.g. high business value).

III. Formal Model and System Implementation

The formal model of our system is based on *management objectives*, which capture the end-to-end performance objectives for services on a given service mesh. These objectives include client requirements and provider priorities.

In our mathematical formulation of management objectives, we denote the services and computing node indexes by i and j, respectively. Further, we let p_{ij} and b_i denote the control actions, where p_{ij} is the routing weight of service i towards node j and b_i is the rate of request blocking for service i. Similarly, l_i denotes the offered load and l_i^c denotes the carried load, which is computed by $l_i^c = l_i(1 - b_i)$. Moreover, u_i denotes the utility of service i (e.g. the business value of service i). Lastly, d_i denotes the response time and O_i denotes the response time objective.

We focus on the following three management objectives.

Management Objective 1 (MO1): the response time of a request of service i is upper bounded by O_i and the overall carried load is maximized:

$$\text{maximize} \sum_i l_i^c \text{ while } d_i < O_i \tag{1}$$

Management Objective 2 (MO2): the response time of a request of service i is upper bounded by O_i and the sum of service utilities is maximized:

$$\text{maximize} \sum_i u_i \text{ while } d_i < O_i \tag{2}$$

Management Objective 3 (MO3): the response time of a request of service i is upper bounded by O_i, the carried load l_i^c of service i is maximized, while service k is prevented from starving (i.e., by specifying a lower threshold l_{min} for service k):

$$\text{maximize } l_i^c \text{ while } d_i < O_i \text{ and } l_k^c > l_{min} \; i \neq k \tag{3}$$

For each management objective, we solve the following optimization problem. Given the offered load of service i, and the response time d_i, we need to find the control parameters p_{ij} and b_i that meet the management objectives. We obtain these parameters through the RL approach shown in Fig. 3.

Following the RL approach, we formalize the above optimization problem as a Markov Decision Process (MDP) $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}^a_{ss'}, \mathcal{R}^a_{ss'}, \gamma, \rho_1, T \rangle$, which is a well-understood model for sequential decision making [7]. The elements of this formalization are defined in our previous work [3].

Our method for solving the MDP and obtaining near-optimal policies includes three procedures. First, we learn a system model using data collected from the testbed. We then use the system model to instantiate a simulator of our testbed. Lastly, we train control policies offline on the simulator through the Proximal Policy Optimization (PPO) RL algorithm and implement the learned policies in the management system.

The management system monitors the testbed through system metrics and runs control policies that perform management actions. The system is written in Bash and Python. The user interface is based on matpotlib and the monitoring pipeline is based on Prometheus and a custom HTTP client. The RL algorithms are implemented using stable-baselines3 and the method for learning the system model is based on Sklearn.

IV. Testbed

To demonstrate our system, we use a testbed at KTH that includes a server cluster connected through a Gigabit Ethernet switch (see Fig. 3). The cluster contains nine Dell PowerEdge R715 2U servers, each with 64 GB RAM, two 12-core AMD Opteron processors, a 500 GB hard disk, and four 1 Gb network interfaces. The tenth machine is a Dell PowerEdge R630 2U with 256 GB RAM, two 12-core Intel Xeon E5-2680 processors, two 1.2 TB hard disks, and twelve 1 Gb network interfaces. All machines run Ubuntu Server 18.04.6 64 bits and their clocks are synchronized through NTP. The orchestration layer and the service mesh are realized using Kubernetes (K8) and Istio.

On top of the Istio service mesh, we implemented two information services, which we call service 1 and service 2. Both services, upon receiving a request with a key, return a corresponding data item from a database. The difference between the services is the structure of data items and the size of the databases.

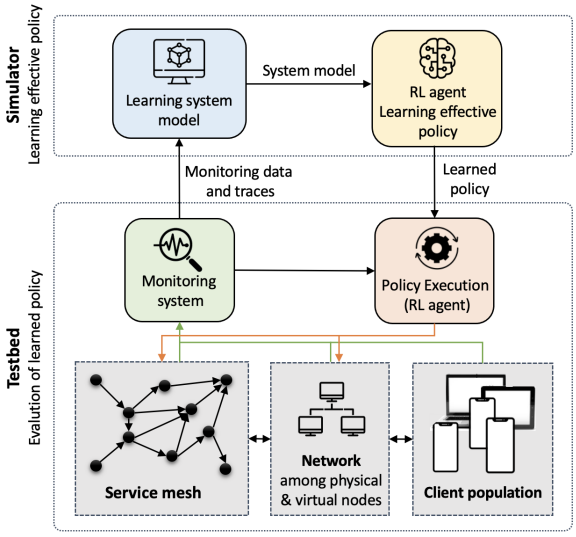

Fig. 3. Our reinforcement learning framework for learning and evaluating control policies of the management system; the KTH testbed implements a service mesh based on Kubernetes and Istio; control policies and system models are learned based on data from the testbed; the management system executes learned control policies; the learned policies take dynamic actions on the testbed based on monitored metrics to meet management objectives.

Figure 1 shows the implementation of the services on the testbed. All nodes are implemented in Python using Flask. The front node provides the web user interface and the other two nodes provide the content for the information service. Each processing node is implemented as a Kubernetes pod.

We implemented a load generator in order to emulate a client population. It is driven by a stochastic process that creates a stream of service requests. We realize two load patterns with this generator.

The *random load pattern* produces a stream of requests at the rate of $l_i(t) \sim \mathbb{U}_{\{5,10,15,20\}}$ requests/second. It changes at every time step to a value drawn uniformly at random from $\{5, 10, 15, 20\}$. A time step is 5 seconds on the testbed.

The *sinusoidal load pattern* produces a stream of requests at the rate of $l_i(t) = 12.5 + 7.5 \times \sin(\frac{2\pi}{T} t + \phi)$ requests/second.

References

[1] S. Alnajdi *et al.*, "A survey on resource allocation in cloud computing," *International Journal on Cloud Computing: Services and Architecture (IJCCSA)*, 2016.

[2] T.-T. Nguyen *et al.*, "Horizontal pod autoscaling in kubernetes for elastic container orchestration," *Sensors*, 2020.

[3] F. S. Samani and R. Stadler, "Dynamically meeting performance objectives for multiple services on a service mesh," in *2022 18th International Conference on Network and Service Management (CNSM)*. IEEE, 2022.

[4] Kubernetes community, "Production-grade container orchestration," 2014. [Online]. Available at: https://kubernetes.io/, Accessed on: June 7, 2022.

[5] Istio community, "Istio homepage," https://istio.io/.

[6] F. Shahab Samani and K. Hammar, "Software framework for a management system based on RL," https://github.com/foroughsh/Framework-for-dynamically-meeting-performanc-objectives, 2022.

[7] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.