Using Reinforcement Learning in Self-driving Systems

DD3359 Course Project
KTH Royal Institute of Technology

Forough Shahab, Kim Hammar

May 27, 2020

1 Motivation

The complexity of modern systems makes the task of operating them difficult. In the last decade we have seen a trend that systems that traditionally would be deployed on a single machine are being distributed to a cluster of machines. By using distributed system architectures, applications can achieve the benefit of increased scalability, fault-tolerance, and availability. However, just as advanced system architectures bring benefits, they also increase complexity of the overall environment, making it hard for human operators to keep up.

In principle, a *self-driving system* is a system that is able to dynamically adjust itself and take operational decisions to meet an objective, reducing the burden on human operators. For instance, a self-driving system could automatically detect that the load on the system is decreasing and, as a consequence, infer that less resources are required for a running application. Figure 1 shows a graphical view of a self-driving system.

In this project, we model the task of finding a control policy for a self-driving system as a reinforcement learning problem. Moreover, we present a proof-of-concept implementation of a self-driving system and demonstrate that the system can learn to allocate resources in a Kubernetes cluster in a dynamic fashion. For our experiments we have utilized a Kubernetes testbed at KTH

2 Testbed

The Kubernetes cluster is deployed on a server rack in our laboratory at KTH. It includes three high-performance machines interconnected by Gigabit Ethernet. These servers are 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. All machines run Ubuntu Server 18.04 64 bits, and their clocks are synchronized through NTP. The cluster includes one master node and two worker nodes which are connected via a switch. We run Kubernetes version v1.17.0 both on the client and the server (see Figure 2(a)).

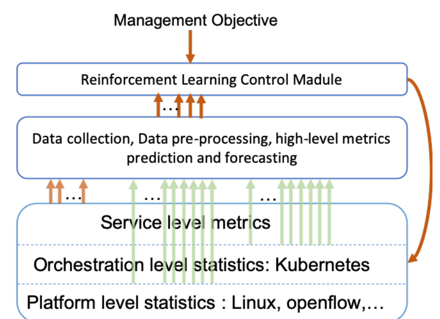
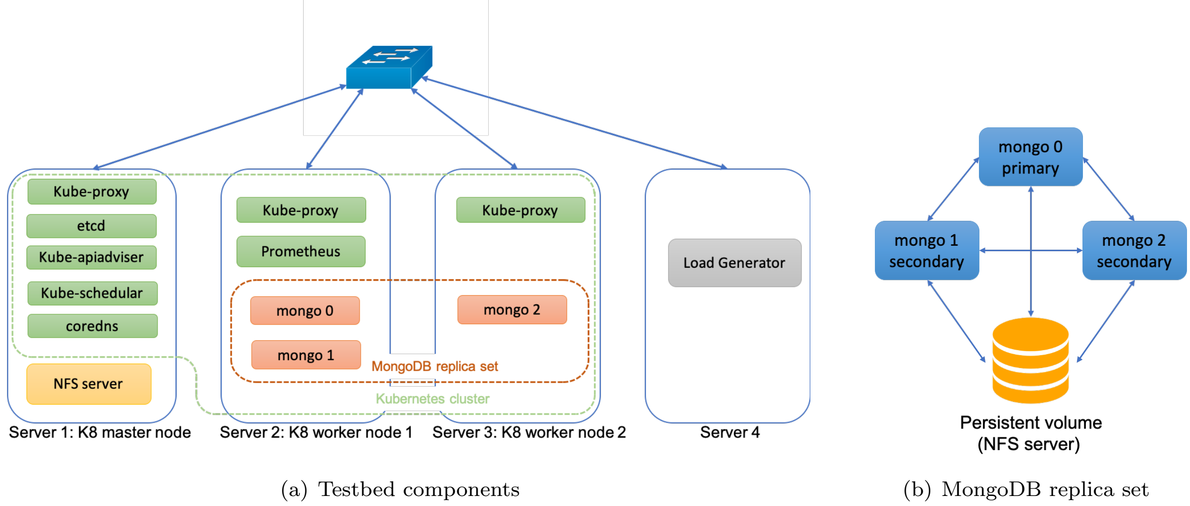


Figure 1: The overview of steps of engineering a self-driving system.

MongoDB version 4.2.6 is running over Kubernetes cluster. We deploy MongoDB replica set with one primary node and two secondary nodes (see Figure 2(b)) [1]. Each of the nodes in the replica set is a MongoDB container. The primary node and one of the secondary nodes are running on the same physical server (worker node 1) and the other secondary node is running on the other. We setup a dynamic provisioning of persistent volumes over network file system (NFS) for the MongoDB replica set. The NFS server is running over the same physical machine on which the master node of the K8 cluster is running.



(a) Testbed components (b) MongoDB replica set

Figure 2: MongoDB replica set over Kubernetes cluster, all the components with blue color are the physical components, K8 pods are shown in green color, MongoDB replica-set is shown with orange color.

3 A Model for Control of Self-Driving Systems

To model the problem we adopt the formal language from the theory of Markov Decision Processes (MDPs) [2] and the notation from [3]. We consider the problem of constructing an optimal control-policy for a self-driving system as a Partially Observed Markov Decision Problem (POMDP) defined by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{P}_{ss'}, \mathcal{R}^a_{ss'}, \gamma \rangle$. The set $\mathcal{S}$ refers to *states* and $\mathcal{A}$ refers to *actions*. The notation $\mathcal{P}^a_{ss'}$ refers to the *transition probability* of moving to state s' when taking action a in state s (Eq. 1) and the notation $\mathcal{R}^a_{ss'}$ refers to the reward when taking an action a in a state s to transition to another state s' (Eq. 2). Effectively, the reward function $\mathcal{R}^a_{ss'}$ is the "task description" of the MDP and specifies the objective that the process seeks to optimize. Finally, $\gamma \in [0, 1]$ is the discount factor, that models the fact that rewards in the near time are valued higher than rewards in the future (assuming $\gamma < 1$). The notation $\mathcal{P}^a_{ss'}, \mathcal{R}^a_{ss'}$, instead of $\mathcal{P}^a_{s_0,s_1,\ldots,s'}, \mathcal{R}^a_{s_0,s_1,\ldots,s'}$ is a reflection of the Markov property; the transition probability and the reward depends only on the current state and action (Eq. 3).

$$\mathcal{P}^a_{ss'} = \mathbb{P}\left[s_{t+1} = s' | s_t = s, a_t = a\right] \qquad \text{Transition probability} \qquad (1)$$

$$\mathcal{R}^a_{ss'} = \mathbb{E}\left[r_{t+1} | a_t = a, s_t = s, s_{t+1} = s'\right] \qquad \text{Expected reward} \qquad (2)$$

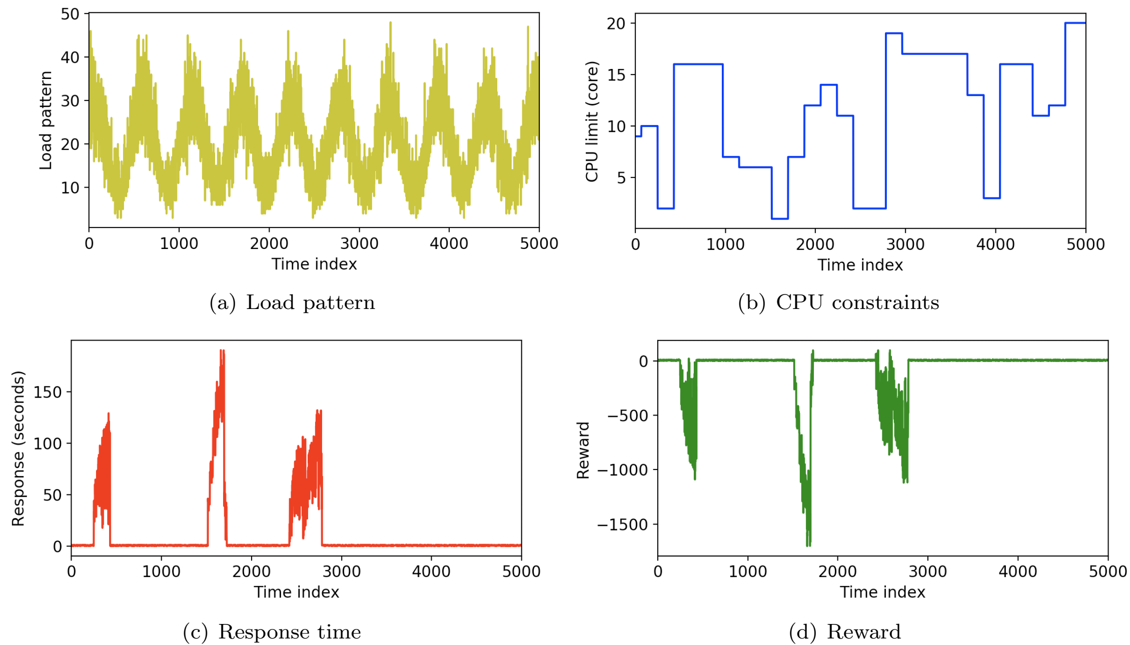
$$\mathbb{P}\left[s_{t+1} | s_t\right] = \mathbb{P}\left[s_{t+1} | s_1, \ldots, s_t\right] \qquad \text{Markov property} \qquad (3)$$

We instantiate the POMDP model for our domain as follows. Let the set of states $\mathcal{S}$ represent the set of Markov states of the testbed described earlier. Next, we define the set of actions, $\mathcal{A}$, to be the set $\{\textsc{NoOp}, \text{CPU} = 1, \ldots, \text{CPU} = 20\}$, representing the CPU allocation for the distributed database in the testbed. Moreover, we use an observation space, $\mathcal{O}$, that consists of all possible combinations of the tuple $\langle o_c, o_{rs} \rangle$, where o_c denotes the current CPU allocation and o_{rs} denotes the observed response time of the database. Finally, we define the reward function to be $\mathcal{R}^a_{ss'} = (0.05 - |g_{rs} - o_{rs}|) * 10$, parameterized by g_{rs} that denotes the desired response time of the database.

4 Results

Using the model described above, we trained an agent using tabular Q-learning with $\gamma = 0.999$, $\alpha = 0.0001$, and $\epsilon = 1$, decayed linearly with a rate of 0.999. Figure 3 visualizes the obtained results after training for 48 hours. As we can see in the Figure, the agent learned to control the CPU constraint based on the measured

response time. Moreover, it is evident that the agent followed the periodic load pattern, which shows that it attempts to adapt to the change of load pattern to have the minimum response time. Furthermore, we can see that after 3000 steps the response time stabilizes despite the face that load pattern changes and we expected change in response time. Finally, we continued the training up to 10 000 steps and observed the same pattern. These results confirm the potential of reinforcement learning to optimize control decisions in network and systems management.



(a) Load pattern

(b) CPU constraints

(c) Response time

(d) Reward

Figure 3: Training results. Figure (a) shows the load pattern of the requests. Figure (b) shows the CPU constraint that agent apply as vertical scaling in the environment. Figure (c) is the response time of the service. Figure (d) shows the calculated reward.

References

[1] MongoDB, “Data model examples and patterns,” https://docs.mongodb.com/manual/tutorial/deploy-replica-set/, 2008.

[2] R. Bellman, “A markovian decision process,” *Journal of Mathematics and Mechanics*, vol. 6, no. 5, pp. 679–684, 1957. [Online]. Available: http://www.jstor.org/stable/24900506

[3] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.