A Reinforcement Learning Based Placement Strategy in Datacenter Networks

Weihong Yang, Yang Qin(✉), and ZhaoZheng Yang

Department of Computer Science, Harbin Institute of Technology (Shenzhen), Shenzhen, China
csyqin@hit.edu.cn

Abstract. As the core infrastructure of cloud computing, the datacenter networks place heavy demands on efficient storage and management of massive data. Data placement strategy, which decides how to assign data to nodes for storage, has a significant impact on the performance of the datacenter. However, most of the existing solutions cannot be better adaptive to the dynamics of the network. Moreover, they focus on where to store the data (i.e., the selection of storage node) but have not considered how to store them (i.e., the selection of routing path). Since reinforcement learning (RL) has been developed as a promising solution to address dynamic network issues, in this paper, we integrate RL into the datacenter networks to deal with the data placement issue. Considering the dynamics of resources, we propose a Q-learning based data placement strategy for datacenter networks. By leveraging Q-learning, each node can adaptively select next-hop based on the network information collected from downstream, and forward the data toward the storage node that has adequate capacity along the path with high available bandwidth. We evaluate our proposal on the NS-3 simulator in terms of average delay, throughput, and load balance. Simulation results show that the Q-learning placement strategy can effectively reduce network delay and increase average throughout while achieving load-balanced among servers.

Keywords: Datacenter networks · Placement strategy · Q-learning

1 Introduction

With the increasing scale of the Internet's users and the emergence of large-scale distributed technologies, the demand for massive data processing has promoted the development of cloud-computing services. As the core infrastructure of cloud computing, Datacenter provides vital support for growing Internet services and applications. There is an urgent need for datacenter networks to deal with massive data under delay constraints [1]. The datacenter networks play a critical role in connecting large and distributed datacenters, providing efficient management and transfer of large amounts of data. Since the centralized placement of data suffers from scalability problems and single point failure, it is necessary to disseminate data across the servers in datacenters. Therefore, how to select suitable storage servers becomes a critical issue.

X. Chu et al. (Eds.): QShine 2019, LNICST 300, pp. 87–101, 2020.
https://doi.org/10.1007/978-3-030-38819-5_6

There are many recent works proposed to deal with the issue of data placement from different perspectives. For example, the random-based placement strategies are simple and popular among several existing systems, such as the Google File System (GFS) [2], Cassandra [3], and the Hadoop Distributed File System (HDFS) [4]. However, random-based placement is not robust to failure and may cause an imbalanced load. On the other hand, some works make placement decisions by considering network resources and fault-tolerance, to reduce the access time and recovery time of data [5–12]. For example, based on the available storage resource, an efficient data placement strategy is proposed in [5] to achieve load balance and satisfy the fault-tolerant requirements. Similar to the existing works mentioned above, we study the data placement issue by considering the available network resource (i.e., bandwidth and storage capacity) in this paper. Moreover, we not only focus on the selection of storage nodes but also design a routing algorithm to find a suitable path to the storage node.

Due to the dynamics of the network, the data placement strategy should be adaptive to the changes in datacenter networks such as the available resource and the occurrence of faults. Reinforcement learning (RL) is agent-based learning that agents can learn by interacting with their environments. RL can explore and learn the dynamics of network and exploit limited resources based on limited knowledge. We leverage the RL method and design a Q-learning based data placement strategy. Each node, including the switch and server, is an agent that can make routing decisions toward the storage node based on the feedback from the datacenter networks.

In this paper, we propose an adaptive Q-learning placement strategy, which consists of the exploration and the exploitation phase. During the exploration phase, each node calculates the Q-value based on the information piggybacked by packets from downstream; and in the exploitation phase, the node makes routing decisions based on the Q-value. The information used for calculating Q-value consists of the available storage capacity of the server and the available bandwidth of the routing path to this server. As a result, the data can be routed along a suitable path toward the storage node. We use the NS-3 simulator to evaluate the performance of our proposal by comparing it with other placement strategies.

The rest of the paper is organized as follows. Section 2 introduces the recent works related to our work. Section 3 presents the model and detailed design of the placement strategy. The simulation results are shown in Sect. 4. Section 5 concludes this work.

2 Related Works

A simple and commonly used placement strategy is random placement, which places the replicas randomly among servers in the datacenter [2, 3, 13]. The random strategy can improve the speed of data repair; however, it is sensitive to multiple and concurrent failures. The rack-aware placement strategy (used in HDFS [4]) places replicas on the servers in both local and remote racks. Each data block is replicated at multiple servers (typically three). In the case of three replicas, one of the replicas is placed on the server of the local rack, and others are placed randomly on servers in the remote rack.

Therefore, it lacks reliability and may cause an imbalance load among the servers of the local and remote rack.

In order to balance and evenly distribute large amounts of the data block, Renuga et al. [5] proposed an efficient data placement and replication scheme to calculate the redundancy parameters accurately that satisfy fault-tolerant requirements. This scheme can improve the utilization of storage space and network bandwidth, reducing data access time and recovery time. Their work focuses on selecting nodes for storage; however, the selection of routing paths to the storage node is not considered. Zaman et al. [6] model the replica placement problem as an optimization problem that minimizing the access time of all servers and objects, given the rate of requests and capacity of servers. Then, a distributed approximation algorithm is proposed to solve the optimization problem. However, the proposed algorithm cannot well suited for the dynamic changes in the datacenter networks.

Rajalakshmi et al. [7] focus on designing an algorithm for optimal replica selection and placement in order to increase the availability of data. The proposed algorithm consists of the file application phase and the replication operation phase. Based on multidimensional locality-preserving mappings, a novel data placement scheme is proposed in [8], which aims at reducing access time of data by supporting dynamic tags. However, it may introduce extra redundancy and has poor scalability in practice. The proposal in [9] is based on the consideration of the availability and popularity of the data. This approach can be tolerant of the occurrence of faults.

AutoPlacer [10] is a self-tuning data placement in a distributed key-value store, which identifies top-k objects that generate most remote operations (i.e., hotspots) for each node of the system, and optimizes the placement of hotspots to minimize the communication between nodes. Lin et al. [11] present a novel placement algorithm that locates the optimal nodes for placing replicas in order to achieve the load balance. Moreover, they propose an algorithm to decide the minimum number of replicas. However, the traffic pattern and locality demands must be known before making the placement decision. Gao et al. [12] addressed the problem of energy cost reduction under both server and network resource constraints within the datacenter and proposed a placement strategy based on ant colony optimization incorporating network resource factors with server resources. CRUSH [14] selects the candidate nodes for storage by using a pseudo-random hashing function. CRUSH can support data placement in a heterogeneous environment; however, the issue of how to route the data toward the storage node is not studied.

3 Q-learning Based Placement

3.1 System Model

We use the Fat-tree network topology [15] as an illustrated example, as shown in Fig. 1. Note that our proposal can be adopted in any topologies of datacenter networks. Assume that the data block $b \in B$ to be placed can arrive at any server. B is the set of

data blocks, and each data block is divided into many data packets. The placement strategy selects a storage server that the available capacity should not be less than m_b, where m_b is the size of data block b. Moreover, the placement strategy finds a suitable forwarding path $p \in P(s,d)$ between server s and d, where $P(s,d)$ is the set of candidate paths between server s and d. The capacity of the path should not be less than r_b, where r_b is the bandwidth requirement of block b. We assume that m_b and r_b are known when the request for placement of data block b arrives. We leverage the reinforcement learning to design distributed placement strategy. More specifically, each node (switch or server) makes a routing decision for each data packet of data block based on the information it learned from the network. We summarize the main notations used in this paper in Table 1.

Table 1. The main notations used in this paper.

Parameter	Definition
b	A data block
B	The set of data block
m_b	The size of data block b
p	A forwarding path
$P(s,i)$	The set of paths from node s to node i
r_b	The bandwidth requirement of block b
S_T^s	The total storage capacity of server s
S_O^s	The used storage capacity of server s
C_T^l	The total bandwidth of link l
C_O^l	The used bandwidth of link l

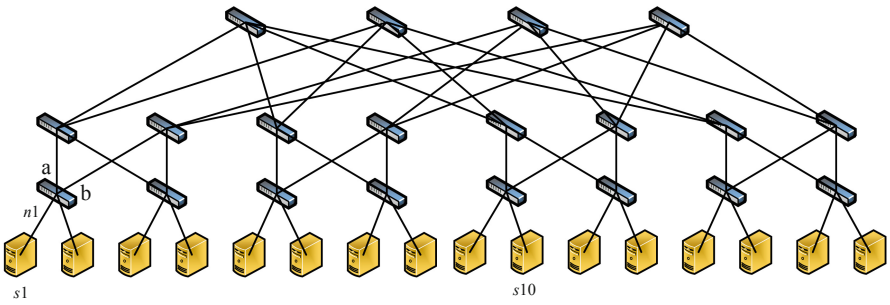

Fig. 1. Fat-tree topology.

3.2 Modeling of Placement Strategy

The goal for each node is to select the next-hop node that jointly maximizes the available storage capacity and bandwidth with constraints. Therefore, we model the placement decision problem for node i when forwarding data block b in datacenter networks as follow:

$$\begin{aligned} \max_{s \in D} \ & \alpha \frac{S_T^s - S_O^s}{S_T^s} + (1-\alpha) \max_{p \in p(i,s)} \min_{l \in p} C_T^l - C_O^l \\ \text{s.t.} \quad & S_T^s - S_O^s \geq m_b \\ & C_T^l - C_O^l \geq r_b \end{aligned} \tag{1}$$

where s denotes the candidate server for storage, $p(i, s)$ is the set of paths from node i to s, and there may exits multiple paths from server i to s. We use p to represent a route between two nodes, p is consists of multiple links. S_T^s is the total storage capacity of the candidate server, S_O^s is the used storage capacity of the candidate server. C_T^l is the bandwidth of link l, and C_O^l is the used bandwidth of link l. The first constraint indicates that the size of the data being placed cannot exceed the remaining capacity of server s. The second constraint indicates that the available bandwidth of every single link must meet the bandwidth requirement of block b. The problem (1) can be interpreted as node i attempts to forward the data block to the next-hop, which has the maximum objective value and satisfies the capacity and bandwidth requirement of the data block. In this hop-by-hop forwarding manner, the data packet can be forwarded toward a suitable storage server along the path with maximum available bandwidth.

3.3 Markov Decision Process and Q-learning

A fundamental assumption based on most reinforcement learning problems is that the interaction process between agent and environment can be regarded as a Markov Decision Process. The Q-learning placement algorithm is based on Markov Decision Process (MDP) for analysis and design [16]. We can use a four-tuple to represent a finite MDP:

$$\langle S_i, A_i, P_i, R_i \rangle_{i=1}^{|N|} \tag{2}$$

S is a state space of node i. At time step t, the state is denoted by $s_i(t) = (\lambda_i^1(t), \ldots, \lambda_i^{|nh|}(t))$, where $\lambda_i^k(t)$ is the information of minimum bandwidth and remaining storage capacity received via interface k of node i, and $|nh|$ is the number of next-hops. A_i is a finite set of actions performed by node i. The action of node i at time step t is $a_i(t)$, which means node i forwards the packet to the selected next-hop at time t. P_i is the transition probability from state s to state s' by performing action a. R_i is the immediate reward received by node i. The reward is defined as the objective function in

optimization problem (1) if the constraints of (1) are not violated. We set the reward to zero if the next-hop cannot meet bandwidth and storage capacity.

The dynamic information of the network (state S) can be learned to make routing and placement decision (action A). The available bandwidth of the routing path and the remaining storage capacity of the server are then used as reward R to train the model.

In order to solve a specific problem, we define an objective function V whose maximum value corresponds to the optimal strategy we want to obtain. We denote a strategy by π. We hope that the system can get the optimal strategy π^* through reinforcement learning, which is a series of action-state transition sequences, which corresponds to the largest converted cumulative return value. Note that there might be more than one value function, so there may be multiple solutions to the problem. The maximum value of the calculated value function can get the optimal strategy. The optimal value function is defined as:

$$V^*(s) = \max_{\pi} E^{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1}\right], \forall s \in S, \tag{3}$$

where $\gamma \in (0, 1)$ is the discount factor, r_t is the reward at time t, $E^{\pi}[\bullet]$ denotes the expectation value under strategy π. According to the dynamic programming, we have the following Bellman equation:

$$V^*(s) = \max_{a}\left[R^a + \gamma E[V^*(s')]\right]. \tag{4}$$

In Q-learning, the Q function is defined as follow:

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a), \forall s \in S, a \in A(s) \tag{5}$$

3.4 Q-learning Based Placement Algorithm

Due to the dynamic changes of the network, the self-optimization of the placement strategy is needed in the datacenter networks. The placement strategy is supposed to learn from the environment and adjust its strategy by trial and error. Q-learning can achieve good learning performance in a complex and dynamic system by learning the optimal policy without prior knowledge. The maximum value of the objective function in (1) can be obtained by the iterative method; therefore, it can be solved by Q-learning based algorithm.

A node in the datacenter networks acts as an agent and learns strategy based on the information piggybacked by the ACK packet from the downstream. The information consists of two parts: the available storage capacity of the server (the first term in the objective function of (1)), and the available bandwidth of downstream path (the second term in the objective function of (1)). The weight α adjusts the importance of bandwidth and storage capacity when making a placement decision.

Updating of Q-value and Stop Condition. Each node i with data packets to send takes action a_i at state s_i based on the strategy $\pi_i(t)$ at the beginning of the time step t. Then, the Q-value for the next time step is updated as follow:

$$\begin{aligned} Q_i^{t+1}(s_i, a_i) &\leftarrow (1-\beta) Q_i^t(s_i, a_i) \\ &+ \beta(\alpha \frac{S_T^s - S_O^s}{S_T^s} + (1-\alpha) \max_{p \in p(i,s)} \min_{l \in p} (C_T^l - C_O^l) + \gamma \max_a Q_i^t(s_i^{t+1}, a)) \end{aligned} \tag{6}$$

The calculation of Q-value stops when the following stop condition is met:

$$\left| Q_i' - Q_i \right| < \frac{\varepsilon(1-\gamma)}{2} \tag{7}$$

where ε is a positive number. If the difference between the two Q-values is small enough, the updating of Q-value stops.

The Detail of Q-learning based Placement. The Q-learning based placement algorithm is presented in Algorithm 1. According to (6), the calculation of Q-value needs the information from the downstream of the routing path. Therefore, we divide the placement strategy into two phases: exploration and exploitation phase.

We augment the ACK with two additional fields: *RemainCapacity* and *MinBandwith*. *RemainCapacity* field is used to record the remaining capacity of the candidate server for storage, and *MinBandwith* records the minimum bandwidth of the routing path. Based on the information in these two fields, a node only forwards the data packet to the next-hop that does not violate the constraints in (1).

Exploration (Line 2–18 in Algorithm 1): In this phase, each node will have to calculate the Q-value locally. Each node sends the request r_q for updating Q-value to its next-hops (Line 3–5). Upon receiving the request r_q, the server responds by sending back the ACK (Line 6–8). Then, the node uses the information piggybacked by the ACKs to update its local Q-values (Line 11). In this hop-by-hop manner, the Q-values along the routing path are updated. The above steps are repeated until the stop condition (7) is met, and then the phase is transited to *exploitation* (Line 12–14). Two conditions can trigger the algorithm to be in the *Exploration* phase: the change of network resources or the initialization of the network. The first condition indicates the algorithm is transited from the Exploitation to the Exploration phase due to the violation of stop condition. In this case, only the node whose Q-value is changed has to issue the request r_q. The second condition indicates that the network is initializing, and there is no Q-value that has been calculated at each node. Therefore, every node has to issue the Q-value updating request r_q.

Exploitation: In this phase, when receiving the request r_d of data block replacement, the switch selects the next-hop p that has the highest Q-value and satisfies the bandwidth and capacity requirement (Line 21–22). The switch also records the next-hop p for this request in the routing table (Line 23), and then the subsequent data packets can be routed to the storage server based on the routing table. When the server receives a placement request r_d, it sends back an ACK tagged with the available capacity (Line 25–27). The algorithm remains in the exploitation phase until the stop condition is violated; then, the phase is transited back to exploration (Line 34–36 and Line 42–44). This condition indicates that the available resource has changed, and the Q-values have to be updated. Note that when a request for data block placement arrives during the exploration phase, the request will still be forwarded to the next-hop that has the highest Q-value, even though the Q-learning based algorithm does not converge.

As a result, the data block will be routed to the storage node along the path that maximizes the objective function in (1), i.e., the selected storage node has maximum available capacity, and the routing path has maximum available bandwidth.

We here use an example to illustrate how the proposed Q-learning based placement works. When the request for data replacement arrives at the server (e.g., server $s1$ in Fig. 1), a request of the data block will be forwarded into the network via switch $n1$. This request carries information about block size m_b and bandwidth requirement r_b. Switch $n1$ will have to decide to which next-hop the request should be forwarded by looking up an Augmented Q-Table (AQT). The main format of AQT is shown in Table 2. Without violating any constraints, the interface with the largest Q-value is selected, and *MinBandwith* is updated by subtracting r_b from the current *MinBandwith*. Moreover, the routing table will be updated: the outgoing interface of the request is recorded. Then, this request is forwarded to the suitable storage server (e.g., server $s10$). Then, $s10$ sends an ACK tagged with current remaining capacity $S_T^{10} - S_O^{10} - m_b$. Upon receiving the ACK, the node updates the local Q-value and *RemainCapacity* value in the AQT. When an ACK reaches server $s1$, the AQTs of all the nodes along the routing path are updated, then the data packet of this data block will be forwarded, taking the same path as the request.

Table 2. Augmented Q-Table.

Interface	*RemainCapacity*	*MinBandwith*	Q-value
a	RC_a	MB_a	Q_a
b	RC_b	MB_b	Q_b

Algorithm 1 Q-learning Placement Algorithm
Input: parameter ε, discount factor γ, learning rate β, data block placement requests r_d, size of data block m_b, bandwidth requirement r_b, the weight α
Output: placement decision

phase ← Exploration;
while phase == Exploration **do**
  for each node i **do**
    Send request r_q for calculating Q-value;
  end for
  while server receives r_q **do**
    Send ACK;
  end while
  while node i receives ACK **do**
    Record RC and MB;
    Updated Q-value by (6);
    if $\left|Q_i'(s_i,a_i)-Q_i(s_i,a_i)\right| < \varepsilon(1-\gamma)/2$ **do**
      phase ← Exploitation;
    end if
      Tag ACK packet with information;
      Send ACK;
  end while
end while
while node *i* receiving placement request r_d do
  if node *i* is switch **do**
    Select next-hop *p* that has the highest Q-value and satisfies the bandwidth and capacity requirements
    Send r_d to the next-hop *p*;
    Add next-hop *p* to routing table;
  else //node *i* is the server
    Calculate available storage capacity;
    Tag ACK packet with available storage capacity;
    Send ACK;
  end if
end while
while node *i* receiving ACK **do**
  Get information from ACK;
  Calculate Q-value by (6);
  Record RC and MB;
  if $\left|Q_i'(s_i,a_i)-Q_i(s_i,a_i)\right| \geq \varepsilon(1-\gamma)/2$ and phase == Exploitation **do**
    phase ← Exploration;

36. **end if;**
37. Send ACK;
38. **end while**
39. **while** node *i* receiving *data* **do**
40. **if** node *i* is switch **do**
41. Send data to the next-hop based on routing table;
42. **if** $\left|Q_i'(s_i,a_i)-Q_i(s_i,a_i)\right| \geq \varepsilon(1-\gamma)/2$ **do**
43. phase ← Exploration;
44. **end if**
45. **else** //node *i* is the server
46. Store the data;
47. **end if**
48. **end while**

4 Simulation

4.1 Setup

We use the NS-3 simulator to perform the evaluation. We adopt Fat-tree as the network topology. The topology is divided into four layers. The upper layer is the core routers, followed by the aggregation routers. The following are edge routers, each of which is directly connected to two server nodes. Each group of aggregation routers, edge routers, and corresponding server nodes is called a Pod. In a k-pod Fat-tree, each pod has two layers of routers (aggregation and edge routers, respectively), each layer has $k/2$ routers, and each edge router connects to $k/2$ servers. We vary the value of k from 4 to 24 in this simulation. The bandwidth of each link is 1000 Mbps. The arrival rate of the data packet is 1 Mbps. The processing delay is 0.001 ms. The size of each data packet is 1 KB. We set the storage capacity to 10 GB for each server. Each data block consists of 1000 data packets. We compare our proposal with the random strategy and CRUSH [14] strategy in terms of delay, throughput, and server load. For the random and CRUSH strategy, we use a modified Nix-Vector algorithm that implemented in NS-3. The discount factor γ is set to 0.7, and the learning rate β is set to 0.5.

4.2 Results and Analysis

Figures 2 and 3 describe the results of the average delay of three placement algorithms. The simulation time is the running time of the simulator. Figure 4 shows the performance of the three algorithms in terms of average throughput. From Figs. 2 and 3, we

can see that when simulation time increases, the Q-learning strategy performs well comparing with the other two strategies. At the beginning of the simulation, the random strategy has a low delay because it does not need to obtain the system information to make the decision. In the meantime, the Q-learning strategy will have to collect information and calculate the Q-value during its exploration phase. In this case, the delay of the Q-learning strategy can be close to or even higher than the random strategy. Later, the Q-learning strategy performs better when the calculation of Q-value converges. CRUSH strategy needs to generate the corresponding distribution function according to the system condition, and the distribution function is fixed. In the data-center where the network traffic changes dynamically, its performance of average delay is not good. The Nix-Vector is an on-demand routing strategy that calculates the routing path on the arrival of requests. To reduce the routing overhead, we record the routing information in the routing table. Then, when the subsequent request arrives, the Nix-Vector will not have to calculate the routing path again. In this case, the delay of random and CRUSH strategy drops over time.

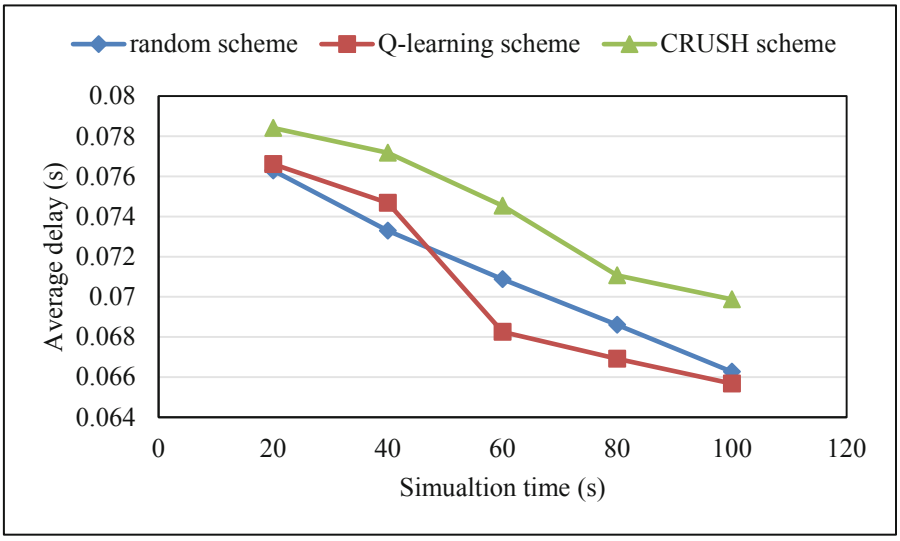

Fig. 2. Average delay vs time ($k = 4$).

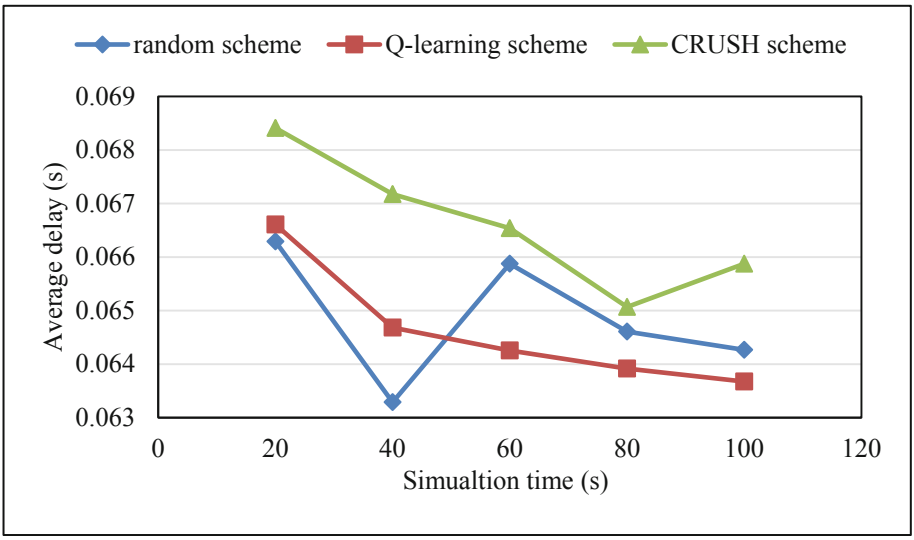

Fig. 3. Average delay vs time (k = 8).

Figure 4 shows the average throughput of different strategies under different values of k. The simulation runs for 100 s. From the figure, we can see that the Q-learning strategy has a slight advantage in terms of average throughput when the system runs long enough. This is because the Q-learning strategy approaches convergence and tends to the optimal solution. During the time of calculating Q-value, we consider the average bandwidth of the path and ensure that loads of each link are balanced. However, the advantage of the Q-learning algorithm in terms of average throughput is not obvious compared with the other two algorithms. This is because the Q-learning algorithm is not optimal in searching for the global optimal Q-value.

In the datacenter networks with unknown network conditions, when the system runs for a short time, the Q-learning method does not fully form a stable Q-value, so the performance of the algorithm on average delay is not good. When the running time is increasing, the Q-learning placement algorithm has a lower average delay because the Q-value in the routing table is gradually stabilized, and the algorithm can be found more accurately.

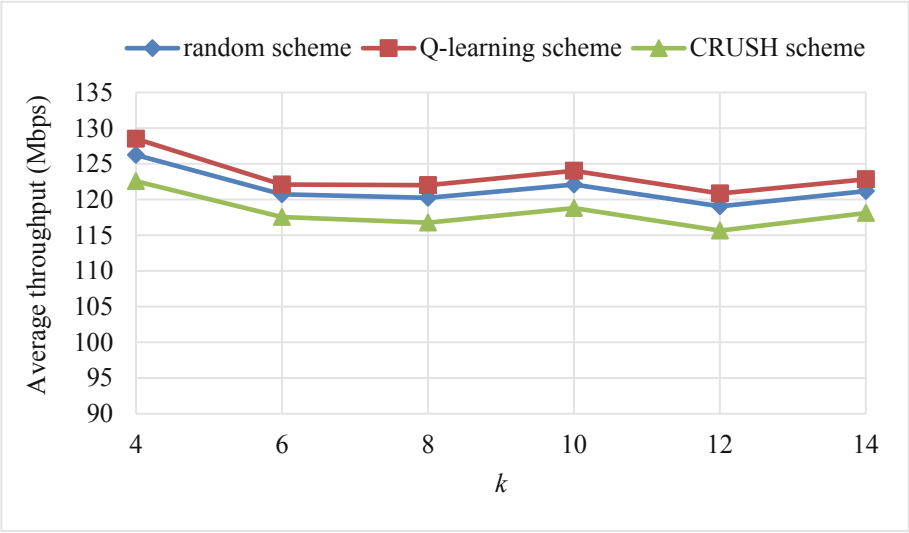

Fig. 4. Average throughput vs. *k*.

Figures 5 and 6 illustrate loads of each server after applying a random strategy and Q-learning strategy, respectively. The initial load of the server node is empty. Then, we apply different placement algorithms to place 10000 data blocks into the network. We set *k* to 8, and there are a total of 432 server nodes.

Random placement strategy randomly selects a server for storing data blocks without assessing the remaining storage capacity, which leads to uneven load among servers and unstable performance. CRUSH strategy generates a hash function based on global network information, but the hash function is relatively fixed once generated. Therefore, its performance is similar to the random placement strategy. The Q-learning strategy considers the remaining storage load of the server and the average bandwidth of path when calculating Q-value; therefore, it can avoid placing data block at a storage node with smaller bandwidth on the selected path.

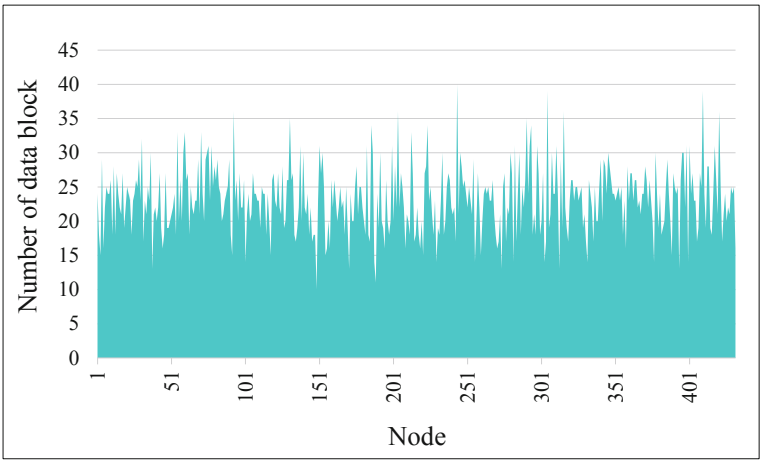

Fig. 5. Load of each node (k = 8, n = 432).

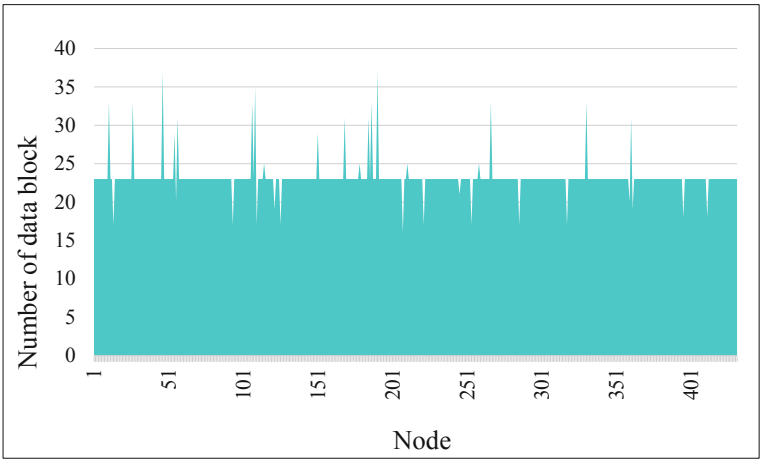

Fig. 6. Load of each node (k = 8, n = 432).

5 Conclusion

In this paper, we propose a reinforcement learning-based data placement strategy for the datacenter networks. The proposed strategy is adaptive to the network dynamic and adjusts the routing and placement decisions based on the available resource. We first model the placement problem as an optimization problem with constraints. Then, we present the detailed design of the Q-learning based data placement strategy, which consists of the exploration and exploitation phase. We perform the simulation to evaluate the performance of our proposal. The simulation results show that the proposed algorithm can effectively balance the load of nodes and improve the average throughput of the system while reducing the average delay.

References

1. Xia, W., Zhao, P., Wen, Y., Xie, H.: A survey on data center networking (DCN): infrastructure and operations. IEEE Commun. Surv. Tutor. **19**, 640–656 (2017). https://doi.org/10.1109/COMST.2016.2626784
2. Ghemawat, S., Gobioff, H., Leung, S.-T.: The Google file system. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, Bolton Landing, NY, pp. 20–43 (2003)
3. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**, 35–40 (2010). https://doi.org/10.1145/1773912.1773922
4. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10 (2010). https://doi.org/10.1109/MSST.2010.5496972
5. Renuga, K., Tan, S.S., Zhu, Y.Q., Low, T.C., Wang, Y.H.: Balanced and efficient data placement and replication strategy for distributed backup storage systems. In: 2009 International Conference on Computational Science and Engineering, pp. 87–94 (2009). https://doi.org/10.1109/CSE.2009.27
6. Zaman, S., Grosu, D.: A distributed algorithm for the replica placement problem. IEEE Trans. Parallel Distrib. Syst. **22**, 1455–1468 (2011). https://doi.org/10.1109/TPDS.2011.27
7. Rajalakshmi, A., Vijayakumar, D., Srinivasagan, K.G.: An improved dynamic data replica selection and placement in cloud. In: 2014 International Conference on Recent Trends in Information Technology, pp. 1–6 (2014). https://doi.org/10.1109/ICRTIT.2014.6996180
8. Vilaça, R., Oliveira, R., Pereira, J.: A correlation-aware data placement strategy for key-value stores. In: Felber, P., Rouvoy, R. (eds.) DAIS 2011. LNCS, vol. 6723, pp. 214–227. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21387-8_17
9. Meroufel, B., Belalem, G.: Dynamic replication based on availability and popularity in the presence of failures. J. Inf. Process. Syst. **8**, 263–278 (2012)
10. Paiva, J., Ruivo, P., Romano, P., Rodrigues, L.: AutoPlacer: scalable self-tuning data placement in distributed key-value stores. ACM Trans. Auton. Adapt. Syst. (TAAS) **9**, 19 (2015)
11. Wu, J.-J., Lin, Y.-F., Liu, P.: Optimal replica placement in hierarchical Data Grids with locality assurance. J. Parallel Distrib. Comput. **68**, 1517–1538 (2008)
12. Gao, C., Wang, H., Zhai, L., Gao, Y., Yi, S.: An energy-aware ant colony algorithm for network-aware virtual machine placement in cloud computing. In: 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS), pp. 669–676. IEEE (2016)
13. Lian, Q., Chen, W., Zhang, Z.: On the impact of replica placement to the reliability of distributed brick storage systems. In: 25th IEEE International Conference on Distributed Computing Systems (ICDCS 2005), pp. 187–196 (2005). https://doi.org/10.1109/ICDCS.2005.56
14. Weil, S.A., Brandt, S.A., Miller, E.L., Maltzahn, C.: CRUSH: controlled, scalable, decentralized placement of replicated data. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC 2006, p. 31 (2006). https://doi.org/10.1109/SC.2006.19
15. Al-Fares, M., Loukissas, A., Vahdat, A.: A scalable, commodity data center network architecture. In: Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, pp. 63–74. ACM, New York (2008). https://doi.org/10.1145/1402958.1402967
16. Doltsinis, S., Ferreira, P., Lohse, N.: An MDP model-based reinforcement learning approach for production station ramp-up optimization: q-learning analysis. IEEE Trans. Syst. Man Cybern.: Syst. **44**, 1125–1138 (2014). https://doi.org/10.1109/TSMC.2013.2294155