Improving Quality of Experience of Service-Chain Deployment for Multiple Users

I-Chih Wang*, Charles H.-P.Wen*, H. Jonathan Chao†
*Department of Electrical and Computer Engineering, National Chiao Tung University (NCTU), Taiwan, R.O.C
†Department of Electrical and Computer Engineering, New York University, Brooklyn, USA
E-mail: *franksum3109@g2.nctu.edu.tw, *opwen@g2.nctu.edu.tw, †chao@nyu.edu

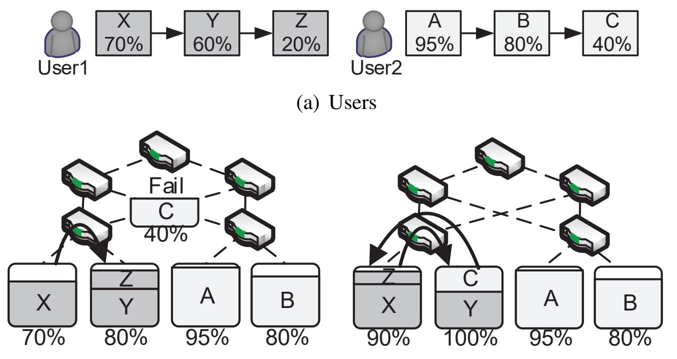
***Abstract*—The fifth generation (5G) mobile communication network aims at providing high-rate, low-latency services. When a user subscribes a chain of service functions (a.k.a. service chain) from the telecom providers, a Service Level Agreement (SLA) is specified according to his requirement. Deploying service chains optimally has always been a big issue. Several previous works have presented various strategies of service-chain deployment for optimizing either latency or computational resources; however, over-optimization of latency or computational resource is not necessarily equivalent to improvement on quality of experience. Therefore, in this paper, we formally formulate this problem of optimizing quality of experience with the queuing theory and mixed-integer linear programming. In addition, we propose an efficient algorithm named "QoE-driven Service-Chain Deployment with Latency Prediction" for deploying a service chain for a user in practice. According to the experiments, our algorithm reduces $> 99\%$ rejections and $> 99\%$ waiting time, notably elevating the quality of experience for users.**

I. Introduction

The fifth generation (5G) is an upcoming network, aiming at providing high-rate, low-latency services. The two key technologies in 5G, network function virtualization (NFV) and software defined networking (SDN), enable the flexibility of the service functions [1]. These service functions are customized for different applications, such as IoT and video streaming. Each application has its own requirement on resource and latency [2]. In general, a user subscribes a sequence of services for an application, and these services are often distributed and visited in a specific order, regarded as a service chain [3]. How to deploy these service chains optimally has always been a big issue.

To deploy a service chain, many factors has to be considered. The most necessary ones are the computational resources, including the CPU usage, the memory usage. Only when these requirements are satisfied can a service function operate normally. Moreover, to connect these service functions, we also need to consider the allocation of network resources, such as the bandwidth allocation and the latency. In 5G network, users have their own demands of these resources according to their customized services. The various demands make it more difficult to deploy service chains efficiently.

In the real world, subscribers of different service chains have different demands on the quality of service. When a user subscribes a service chain, a **Service-Level Agreement** (SLA) is committed by the user and the telecom providers.



(a) Users

(b) Case 1 (c) Case 2

Fig. 1: Two cases to illustrate how users occupy the resource and how they affect each other.

The SLA specifies the resource and the latency guaranteed by the telecom providers. Therefore, according to the SLAs, we can learn users' requirements and analyze how users should occupy the resource. Our goal is to satisfy every user's requirement and to optimize the deployment of service chains for as many users as possible.

Previous studies have proposed many deployment strategies to optimize the resource allocation. However, optimizing the resource allocation is not necessarily equivalent to optimizing the quality of experience. The SLAs of low-requirement users may be overly satisfied, and SLAs of high-requirement users may be violated. Two cases in Figure. 1 are used to illustrate this fact, where a 1-2-2 fat-tree topology is applied to observe the CPU usages in percentages.

(Case 1) Over satisfaction of latency: In this case, we assume that the deployment attempts to minimize the latency. In the beginning, User 1 in Figure. 1(a) first arrives and subscribes some resource for a service chain (X, Y, Z) with latency requirement (< 5 hops). The service chain is deployed as the left-hand side in Figure. 1(b) to minimize the latency. Here, the latency of User 1's service chain is 2, which is lower than the latency requirement (< 5 hops). After that, User 2 in Figure. 1(a) subscribes some resource for his own service chain (A, B, C). However, User 2's service chain is failed to be deployed, because there is no enough CPU resource for service C. The deploying result is shown at the right-hand side of Figure. 1(b). In this case, the latency requirement of User 1 is over satisfied, leading to the failure of User 2. Only User

1 is satisfied.

(Case 2) Over allocation of CPU usage: In this case, we assume that the deployment attempts to optimize resource allocation. In the beginning, User 1 in Figure. 1(a) first arrives and subscribes some resource for a service chain (X, Y, Z) with a latency requirement ($<$ 3 hops). The service chain is deployed as the left-hand side in Figure. 1(c) to free more CPU usage for the next user. Here, the latency of User 1's service chain is 4, which violates the latency requirement ($<$ 3 hops). After that, User 2 in Figure. 1(a) subscribes resources for his own service chain (A, B, C). As the deploying result shown in Figure. 1(c), User 2's service chain can be deployed. In this case, User 2 is satisfied, yet User 1's latency requirement is violated. Only User 2 is satisfied.

Such two cases stated above distinguish the optimization of the resource allocation and the optimization of quality of experience. The previous works [4]–[13] cannot directly be applied to optimize the quality of experience. Obviously, a new model considering users' requirement needs to be proposed to analyze the quality of experience. In this paper, the Erlang-C queuing model is used for analyzing performance of the service-chain deployment. We observe how much time a user will wait to be satisfied, and use the waiting time of all users as the criteria of quality of experience.

Considering the quality of experience, service-chain deployment becomes a multi-objective NP-hard problem. We need to optimize the usage of computational resource and network resource; also, we need to satisfy every user's SLA. Moreover, because users arrive at different time, the deployment for the early-arrival users can affect the deployment of the late-arrival ones. Considering the complexity of the problem, we propose a heuristic algorithm for service-chain deployment targeting multiple users.

Our heuristic algorithm, named ***"QoE-driven Service-Chain Deployment with Latency Prediction" (QoEDD)***, is inspired from the observation of the two cases in Figure. 1: overly satisfying latency and overly allocating computational resources. To deal with these two problems, our algorithm attempts to take the advantages of both latency-driven deployment (LDD) and resource-driven deployment (RDD) by making choices between them for improving the quality of experience. We propose a predictive method that can efficiently predict the latency of resource-driven deployment without any physical deployment. With the predicted latency, we can decide whether resource-driven or latency-driven deployment should be applied.

To evaluate our algorithm, we measure the rejections to users to observe the number of users who are failed to be satisfied. 99.85% of rejections to users is reduced by QoEDD, comparing to RDD and LDD. $> 99\%$ of waiting time is reduced with QoEDD. These results illustrate that optimizing latency of computational resource does not necessarily improve the quality of experience. Moreover, QoEDD is also compared to the latest algorithm SOVWin [13], which also targets at QoE. SOVWin rejects users 80,875 times and the users' waiting time is 157. The rejection is 909-times larger than QoEDD and the waiting time is 15700-times larger than the users with QoEDD. These results show that QoEDD notably elevates the quality of experience of users.

II. Related Works

Deploying service chains has always been a big issue since NFV enabled the flexibility of network-function placement. Some placement mechanism has been proposed to optimize the energy usage or to optimize the computational resources ([4]–[6]). However, network resources should also be considered. Network resource allocation is another important factor that affects the quality of service. G. Wang and T. E. Ng (2010) informed that notable throughput instability and delay can be caused by virtualization, even though the traffic in datacenter network is light [7]. Some placement mechanisms have been proposed to optimize the allocation of both computational resource and network resources ([8]–[10]). R. Wang et al. (2014) proposed MAPLE to balance computational resource and network resource [8]. T. Ma et al. (2017) leveraged Markov approximation [9]. K. Yang et al. (2016) proposed Merge-RD to reduce both energy consumption and transmission delay [10]. These solutions optimize both the computational resource and network resource. However, as the two examples in Figure. 1 shows, optimizing the resource allocation does not necessary improve the quality of experience.

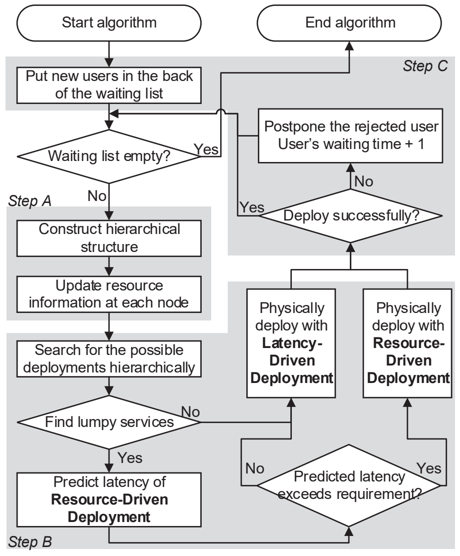
To optimizing the quality of experience, we have to understand users' needs through SLAs and avoid any violation. Placement strategies has been proposed targeting SLAs ([11], [13]). K. Katsalis et al. (2016) proposed LBVS for VM scheduling with SLA considered [11]. Y.-M. Pai et al. (2013) proposed a VM deployment algorithm SOVWin [13], using a searching window to restrict the searching space for limiting the latency. These studies evaluated performance by observing how many users were rejected by the system or how many SLAs were violated. However, long waiting time could happen to a user, because these algorithm may postpone a request to benefit the other users.

The goal of this paper is to maximize the number of satisfied users and also minimize the waiting time of users. We proposed an algorithm to optimize the quality of experience before any service function migration, and observe its performance with random-arriving users.

III. Modeling The Service Chain Deployment Problem with QoE Considered

Queuing theory describes users' experience with waiting time, and observes a system in continuous time with multiple users. We can model the service capacity for service chains with a queuing model. In this study, we leverage the Erlang-C model to quantify the performance of a deploying mechanism with users' waiting times, and compare the qualities of experience of different service-chain deploying strategies.

In Erlang-C model (or M/M/c queue), **c** is defined as the number of the agents in the system or the maximal number of users that a system can serve at one time. With given server capacities ($Serv$) and service function (SF) requests, we map



Start algorithm
End algorithm
Step C
Put new users in the back of the waiting list
Postpone the rejected user User's waiting time + 1
Waiting list empty?
Yes
No
Step A
No
Yes
Deploy successfully?
Construct hierarchical structure
Update resource information at each node
Physically deploy with Latency-Driven Deployment
Physically deploy with Resource-Driven Deployment
Search for the possible deployments hierarchically
Find lumpy services
No
Yes
Predict latency of Resource-Driven Deployment
No
Predicted latency exceeds requirement?
Yes
Step B

Fig. 2: The flow of our algorithm "QoE-driven Service-Chain Deployment with Latency Prediction".

the ideal service capacity of a topology to the parameter **c** with this formula:

$$c = min\left(\frac{\sum Serv_i^{CPU}}{\sum SF_j^{CPU}}, \frac{\sum Serv_i^{Mem}}{\sum SF_j^{Mem}}, \frac{\sum Serv_i^{BW}}{\sum SF_j^{BW}}\right) \quad (1)$$

Yet, the real service capacity is less than the ideal one due to latency dissatisfaction and capacity waist cased by leftover space on the servers. We denote the real service capacity as $c' = \varphi c$, where φ is a loss factor between $(0, 1]$. Our goal is to maximize φ for greater c', which leads to lower waiting times and improves the quality of experience.

Erlang-C model with c' is not enough for modeling our system, because users' SLAs are independent and cannot be directly transformed to c'. Therefore, we need mixed integer linear programming (MILP) to maximize c'. According to SLAs, servers must provide enough CPU, memory, bandwidth and satisfy the latency requirement (constraints). Then, we can maximize the satisfied users by minimizing the number of rejected users (objective function):

$$minimize\left(\sum_{i=0}^{User^N} Rej_i\right),\ Rej_i = \begin{cases}0, accepted\\ 1, rejected\end{cases} \quad (2)$$

Then, c' can be derived. However, due to the complexity, the MILP cannot solve the problem efficiently. Thus, we propose a heuristic algorithm, "***QoE-driven Service-Chain Deployment with Latency Prediction***", for practical service-chain deployment.

IV. QoE-driven Service-Chain Deployment with Latency Prediction

In this section, we will introduce our heuristic algorithm in details. From the observation of the two cases in Figure. 1, we can learn that neither only optimizing the latency nor only optimizing the resource can necessarily improve the quality of experience. In the two cases, the only difference in their assumptions is User 1's latency requirement. We call the two strategies, that target different objectives, "**resource-driven deployment**"(RDD) and "**latency-driven deployment**"(LDD). Resource-driven deployment can fully utilize the fragmented resources; however, the scattered service functions cause high latency. Latency-driven deployment can easily fulfill users' latency requirement; however, the fragmented resources cannot be fully utilized. Therefore, choosing a proper deploying strategy according to users' demands is the main feature of this algorithm. We propose an efficient method to predict whether a user's latency requirement can be satisfied without physical deployment. Furthermore, we accelerate our algorithm with a hierarchical searching. The spanning tree protocol is first applied to build a multi-level tree structure. On each level, our algorithm performs the decision making and the resource searching. In short, our algorithm "QoE-driven Service-Chain Deployment with Latency Prediction" mainly consists of three steps: (a) building a hierarchical structure, (b) performing latency prediction and deployment, and (c) handling failures of deployment. The flow of the algorithm is illustrated in Figure. 2, in which the gray blocks correspond to the three stages, respectively.

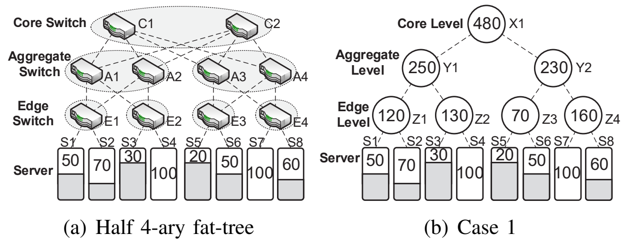


Fig. 3: A half 4-ary fat-tree for illustrating the building of a hierarchical structure.

A. Building a Hierarchical Structure

First, we use STP to get a hierarchical structure and collect the resource information hierarchically. Here, we use a fat-tree topology as an example. The network in Figure. 3(a) is a half 4-ary fat-tree topology. After STP is performed on the topology, we can acquire the spanning tree in Figure. 3(b). With the spanning tree, we can collect the resource information hierarchically on each nodes.

To simplify the computation of multi-dimensional resources, we use L^1-Norm to transform the normalized CPU capacity, memory capacity and bandwidth capacity into a single dimension. We redefine the new representation of resources as "score". We use the resource information "score" to find out if the resource is available. For example, in Figure. 3(b), the server S1 has the score 50 and the server S2 has the score 70. If we observe the resource information from a higher level, the edge-level node Z1 has the score 50 (S1) + 70 (S2) = 120. Similarly, the aggregate-level node Y1 has the score 250, and the core-level node X1 has the score 480. When this step is done, a hierarchical structure of resource information is constructed.

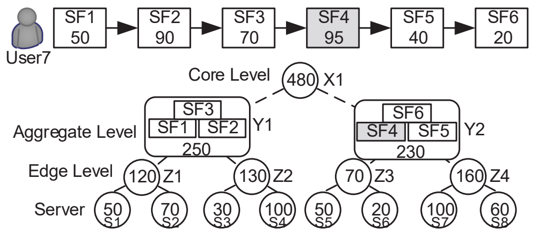


Fig. 4: The flow of our algorithm "A deploying example illustrating the condition when a lumpy SF appears".

With the hierarchical structure of resource information, we can perform the latency prediction and resource searching hierarchically. For instance, in Figure. 3(b), if we try to deploy a service function with a score 40, we will start the searching from the core level. At the core-level node X1, our algorithm will choose between Y1 and Y2 according to the result of algorithm in Section IV-B. In the real implementation, we also record the maximal capacity of each resource at the upper level. The record is for making sure that the three resource in SLAs, which we simplify with L^1-Norm, can all be satisfied. By recording the three maximal capacities, we can make sure that the three requirements can all be satisfied.

B. Performing Latency Prediction and Deployment

After we build the hierarchical structure with resource information, we can perform the latency prediction and deployment at each level. However, if we look closer to the two cases in Figure. 1, we can see that the latency prediction only needs to be performed under certain conditions. The difference between the two cases in Figure. 1 is how User 1 leaves the unused resource. In case 1, User 1 leaves 30% and 20% CPU capacities for User 2; in case 2, User 1 leaves 10% and 40% CPU capacities for User 2. Moreover, User 2's service function C requires 40% CPU usage. The special service function C can be deployed in case 2 but not in case 1. This example points out the special requirement of service function C that differentiates the results of the two cases. Therefore, the latency prediction only needs to be performed only when we find the special service function such as C; otherwise, a service can be deployed in both cases or rejected in both cases.

Hence, we leverage this special condition to reduce the number of prediction for saving computational costs. In the example, we name the special service function "***lumpy service function***" (***Lumpy SF***). The resource requirements of Lumpy SFs is the main cause of the different outcomes between the two cases in Figure. 1. We define that the resource requirement of Lumpy SFs must satisfy all the following conditions: (1) the service must have the largest requirement of score in i-th user's service chain. The largest-requirement service function is the hardest one to deploy and is the key effect of the deployment. (2) the service functions before the Lumpy SF can be deployed into the i-th server (or node). (3) However, the Lumpy SF cannot be deployed into the i-th server (or node) due to the previous deployment. Only when a Lumpy SF is detected will the algorithm trigger the latency prediction.

An example is given in Figure. 4 to illustrate a Lumpy SF. In this example, we need to deploy User 7's service chain.

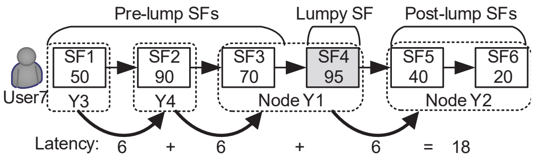


Fig. 5: Latency prediction example.

With the hierarchical structure, we start the searching from the core-level node X1. At the node X1, two candidate nodes (Y1, Y2) can be chosen for deploying service functions. In this case in Figure. 4, if service functions (SF1, SF2, SF3) are deployed in the subtree Y1, then the remained score of Y1 is 250-50(SF1)-90(SF2)-70(SF3) = 40. The remained score of Y1 40 is not enough for the service function SF4. Hence, the service function SF4, which requires the highest score and failed to be put into Y1, is a Lumpy SF. Then, the algorithm triggers the latency prediction and chooses between resource-driven deployment and latency-driven deployment.

We predict the latency of the resource-driven deployment to see whether user's latency requirement allows us to optimize the resources in first place; otherwise, we apply the latency-driven deployment. In the example in Figure. 4, we decide whether User 7's latency requirement allow us to rearrange service functions to make a complete resource for the Lumpy SF. To predict the latency of resource-driven deployment, we need to first divide a service chain into three parts: pre-lump SFs, lumpy SF, and post-lump SFs. For example, User 7's service chain is divided into three parts: pre-lump SFs (SF1, SF2, SF3), Lumpy SF (SF4), and post-lump SFs (SF5, SF6) as Figure. 5 shows. The algorithm predicts the latency in these three parts separately.

For the pre-lump SFs, we first calculate how many service functions need to be removed and leave out enough resources for the Lumpy SF to fit into a server (or node). Because we are predicting the latency of resource-driven deployment, we assume that each removed service function is assigned to a single server (or node) for utilizing the fragmented resource on each server (or node). The number (N_{rm}) of the removed service functions can be obtained with the following formula:

$$N_{rm} = min\{idx - 1 | \sum_{j=idx}^{LSF-1} SFj^{score} \leq Node_k^{score}\} \quad (3)$$

For example, in Figure. 4, we calculate how many service functions need to be remove to fit the Lumpy SF (SF4) into the node Y1. In this case, the service functions (SF1, SF2) need to be removed (70 (SF3) + 95 ((SF4)) = 165 < 250). Then, the latency (hops) of pre-lump SFs can be predicted as:

$$Latency_{pre-lump} = (N_{removed} - 1) * level * 2 \quad (4)$$

Thus, the pre-lump latency of User 7 is $(2-1)*3*2 = 6$ hops, because the algorithm is so far searching at the core-level node X1.

For the latency beside the Lumpy SF, because a single server (or node) is required for the Lumpy Service, the latency can be obtained as the following:

$$Latency_{lump} = 2 * level * 2 \quad (5)$$

As for the post-lump SFs, considering that the latency agreement might be tight after deploying the Lumpy SF, we simply use the latency-driven deployment on post-lump SFs to get the latency. Therefore, the overall predicted latency at the aggregate level is the sum of these latencies:

$$Latency = Latency_{pre} + Latency_{lump} + Latency_{post} \quad (6)$$

The latency prediction of User 7 is given in Figure. 5. According to the previous description, we first assumed that SF1 and SF2 are placed on nodes (Y3, Y4). Then, we calculate the latency beside the Lumpy SF and obtain the post-lump latency from the case in Figure. 4. In short, the pre-lump latency is 6 hops. The latency beside the Lumpy SF is 12 hops, and the post-lump latency is 0 hops. Therefore, the latency prediction of User 7 on the aggregate level is 18 hops. Adding up the latency prediction at each level, we can obtain the overall latency prediction. If User 7's latency requirement is loose, then we will apply resource-driven deployment to let SF1 and SF2 utilize the fragmented resource on Y3 and Y4; otherwise, if User 7's latency requirement is strict, then the latency-driven deployment will be applied.

C. Handling Failures of Deployment

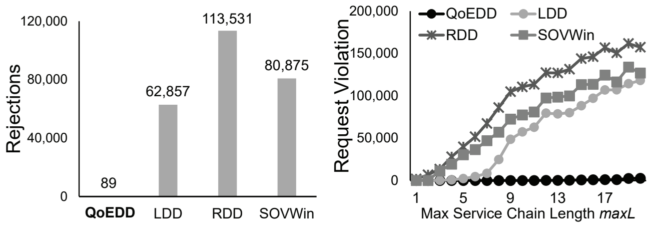
In some cases, we cannot deploy users' service chains, because no feasible deployment is found. When no deployment can satisfy user's requirement, we postpone the users' request. Those postponed users will be put into a queue and wait for a retry. Our next problem is how to rearrange these postponed users with new-coming users. We have tried several methods including reserving or sorting, and found out that the one without sorting and reserving cases the less waiting time. Moreover, giving postponed users higher priority than new-arriving users can also decrease the waiting time in average.

V. Performance Evaluation

We create a network simulator with C++ language to simulate a real network, and choose k-ary fat-tree topologies for the evaluation. Because our goal is to improve users' experience, we observe users' waiting time and rejections to users.We claim that overly optimizing the resources or the latency is not necessary the same as optimizing the quality of experience. To support the claim with demonstrations, we will compare our algorithm with pure **Resource-Driven Deployment (RDD)** and pure **Latency-Driven Deployment (LDD)**. Here, we use the two methods, which are mentioned in the paper [13], **Ordered First-Fit** for LDD and **Best-Fit** for RDD. Moreover, we compare our algorithm with the latest one SOVWin [13], which also optimizes the service-chain deployment considering users' SLAs, to evaluate the performance of our algorithm.

A. Experimental Setting

Our evaluations are mainly on a 6-ary fat-tree network with 54 servers. Each server has CPU capacity noted as 100% and memory capacities noted as 100%. The bandwidth of every link is set to 1024 Mbps. In the experiments, we simulate users' arrivals and departures on a time line with 1000 time steps. The users' arrival rate is $\lambda = 0.4$ and users' service rate is $\mu = 0.1$. To compare to the ideal case of a queuing model, we simulate users' arrivals with a Poisson process and simulate users' service duration with an exponential distribution. Each user has a service chain with a latency requirement and several service functions. The number of the service functions is uniformly distributed from 1 to **maxL**. Each service function requires 5% to 100% CPU capacity and 5% to 100% memory capacity. The CPU and memory requirement is uniformly distributed and the minimum 5% represents the base load of a service function. Moreover, users subscribe some bandwidth for their service chains. The bandwidth requirement is normally distributed with the mean 100 Mbps and the variance 20 Mbps. With the users and the topology, we can start to evaluate the performance of our algorithm.



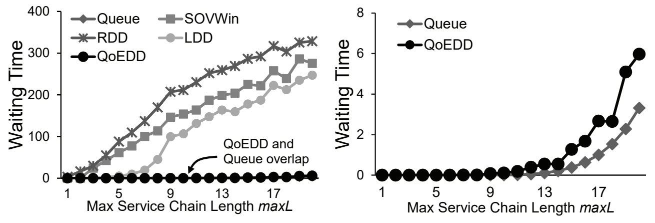
(a) maxL = 11 (b) maxL = 1 to 20

Fig. 6: Number of rejections of users

B. Comparison on Rejections to Users

When there is no feasible deployment to meet a user's requirement, the deploying failure is recorded and denoted as a rejection to a user. A rejected user is postponed and will retry. In some cases, a user might be rejected for several times, and all of these rejections are recorded. The rejections are shown in Figure. 6. In Figure. 6(a), we fix the maximum length of service chains (maxL =11) and compare the rejections between each deployment strategies. Our algorithm only rejects users 89 times within the experiment, while RDD and LDD reject users for more than 60,000 times. The results suggest that overly optimizing the resources or the latency for a single user does not reduce the number of rejections, and even causes more rejections. On the other hand, SOVWin rejects users 80,875 times, which is much larger than our algorithm. Our algorithm rejects only $< 0.14\%$ of the requests of either RDD, LDD or SOVWin.

Next, we would like to show how the number of rejections grows when the length of service chain increases. We scale the maximum length of service chains maxL from 1 to 20 and observe the rejections. In Figure. 6(b), our algorithm rejects users for less than 10,000 times for any maxL within 20. The rejections of RDD and SOVWin start to increase significantly for maxL = 3, and the rejections of LDD start to increase for maxL = 7. This result shows that even for long service chains, our algorithm can still outperform the others.



(a) Different deployment strategies (b) QoEDD and Queuing model

Fig. 7: Waiting time of users

C. Comparison on Waiting Time

When a rejection occurs, the rejected user is postponed. We record the waiting time of each user and observe the distribution of the waiting time, reflecting the quality of experience. According to section III-B, we show that lower waiting time leads to the lower loss factor. This means lower waiting time indicates better system utilization. The waiting time is shown in Table III. Comparing to RDD and LDD, our algorithm significantly reduces the waiting time by 99.83% and 99.93%, respectively. Moreover, comparing to SOVWin, our algorithm reduces 99.4% of users' waiting time.

On the other hand, the variance of waiting time indicates the fairness between users. Every user should be treated equally for achieving good quality of experience. Table III also shows the variance of waiting time. If we apply our algorithm, the variance of the waiting time is 0.16, while RDD is 237 and LDD is 268. Overly satisfying the latency or overly allocating resource only benefits to some users, while other users might wait for a long time. For SOVWin, the variance of waiting time is 96. One reason that SOVWin and our algorithm perform better than RDD and LDD is that we consider users' SLA and improves the quality of experience.

Next, we would like to know how the waiting time grows when the length of service chain increases. We scale the maximum length of service chains maxL from 1 to 20 and observe the mean of users' waiting time. Here, with the queuing model, we can calculate the ideal case of resource allocation. The waiting time according to each maxL is shown in Figure. 7(a). Our algorithm has the minimum of waiting time comparing to RDD, LDD or SOVWin for any maxL within 20. In Figure. 7(a), the lines of our algorithm and the queuing model are overlapped, so we scale up to observe the difference. In Figure. 7(b), the waiting time of our algorithm is larger than the one of queuing model when maxL is larger than 9. The reason is that the queuing model sums all the resources and obtains the number of agent c, so the fragmented resources will not happen in the queuing model. Moreover, the queuing model does not consider the latency requirement, so users will not be rejected due to the latency requirement. Even so, the queuing model provides the lower bound of the waiting time.

VI. Conclusion

In this paper, the problem of optimizing quality of experience is formulated formally with the queuing theory and mixed-integer linear programming. We then propose a heuristic algorithm named "QoE-driven Service-Chain Deployment with Latency Prediction" (QoEDD) for deploying service chains in practice. According to our experiments, QoEDD reduces more than 99% rejections and more than 99% of waiting time, comparing to RDD and LDD. These results illustrate that optimizing latency or computational resource does not necessarily improve the quality of experience. Moreover, QoEDD reduces 99.89% rejections and 99.99% waiting time, comparing to SOVWin. In conclusion, QoEDD is an efficient algorithm for service-chain deployment which notably elevate the quality of experience.

Waiting time	RDD	LDD	SOVWin	**QoEDD**
Mean	195	89	157	**0.01**
Variance	237	96	268	**0.16**

TABLE I: Statistical comparison on waiting time of users.

Acknowledgment

This paper is particularly supported by "Aiming for the SPROUT Project-Center for Open Intelligent Connectivity" of National Chiao Tung University and Ministry of Education, Taiwan, R.O.C.

References

[1] J. G. Andrews, S. Buzzi, W. Choi, S. V. Hanly, A. Lozano, A. C. K. Soong, and J. C. Zhang, "What will 5g be?" *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 6, pp. 1065–1082, June 2014.

[2] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing—a key technology towards 5g," *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.

[3] S. Mehraghdam, M. Keller, and H. Karl, "Specifying and placing chains of virtual network functions," in *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, Oct 2014, pp. 7–13.

[4] L. Chen and H. Shen, "Consolidating complementary vms with spatial/temporal-awareness in cloud datacenters," in *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, April 2014, pp. 1033–1041.

[5] F. Farahnakian, P. Liljeberg, T. Pahikkala, J. Plosila, and H. Tenhunen, "Hierarchical vm management architecture for cloud data centers," in *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, Dec 2014, pp. 306–311.

[6] A. Khosravi, L. L. H. Andrew, and R. Buyya, "Dynamic vm placement method for minimizing energy and carbon cost in geographically distributed cloud data centers," *IEEE Transactions on Sustainable Computing*, vol. 2, no. 2, pp. 183–196, April 2017.

[7] G. Wang and T. S. E. Ng, "The impact of virtualization on network performance of amazon ec2 data center," in *2010 Proceedings IEEE INFOCOM*, March 2010, pp. 1–9.

[8] R. Wang, R. Esteves, L. Shi, J. A. Wickboldt, B. Jennings, and L. Z. Granville, "Network-aware placement of virtual machine ensembles using effective bandwidth estimation," in *10th International Conference on Network and Service Management (CNSM) and Workshop*, Nov 2014, pp. 100–108.

[9] T. Ma, J. Wu, Y. Hu, and W. Huang, "Optimal vm placement for traffic scalability using markov chain in cloud data centre networks," *Electronics Letters*, vol. 53, no. 9, pp. 602–604, 2017.

[10] K. Yang, H. Zhang, and P. Hong, "Energy-aware service function placement for service function chaining in data centers," in *2016 IEEE Global Communications Conference (GLOBECOM)*, Dec 2016, pp. 1–6.

[11] K. Katsalis, T. G. Papaioannou, N. Nikaein, and L. Tassiulas, "Sla-driven vm scheduling in mobile edge computing," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, June 2016, pp. 750–757.

[12] K. Lu, R. Yahyapour, P. Wieder, C. Kotsokalis, E. Yaqub, and A. I. Jehangiri, "Qos-aware vm placement in multi-domain service level agreements scenarios," in *2013 IEEE Sixth International Conference on Cloud Computing*, June 2013, pp. 661–668.

[13] Y.-M. Pai, C. H. P. Wen, and L.-P. Tung, "Sla-driven ordered variable-width windowing for service-chain deployment in sdn datacenters," in *2017 International Conference on Information Networking (ICOIN)*, Jan 2017, pp. 167–172.