Hindawi
Wireless Communications and Mobile Computing
Volume 2020, Article ID 9869434, 15 pages
https://doi.org/10.1155/2020/9869434

Research Article

Multiaccess Edge Computing-Based Simulation as a Service for 5G Mobile Applications: A Case Study of Tollgate Selection for Autonomous Vehicles

Junhee Lee, Sungjoo Kang, Jaeho Jeon, and Ingeol Chun

Electronics and Telecommunications Research Institute (ETRI), Daejeon, Republic of Korea

Correspondence should be addressed to Ingeol Chun; igchun@etri.re.kr

Received 7 June 2019; Revised 23 October 2019; Accepted 21 November 2019; Published 7 March 2020

Academic Editor: Donatella Darsena

As the data rate and area capacity are enormously increased with the advent of 5G wireless communication, the network latency becomes a severe issue in a 5G network. Since there are various types of terminals in a 5G network such as vehicles, medical devices, robots, drones, and various sensors which perform complex tasks interacting with other devices dynamically, there is a need to handle heavy computing resource intensive operations. Placing a multiaccess edge computing (MEC) server at the base station, which is located at the edge, can be one of the solutions to it. The application running on the MEC platform needs a specific simulation technique to analyze complex systems inside the MEC network. We proposed and implemented a simulation as a service (SIMaaS) for the MEC platform, which is to offload the simulation using a Cloud infrastructure based on the concept of computation offloading. In the case study, the Monte-Carlo simulations are conducted using the proposed SIMaaS to select the optimal highway tollgate where vehicles are allowed to enter. It shows how clients of the MEC platform use SIMaaS to obtain certain goals.

1. Introduction

With the advent of the 5G wireless communication service, the aggregate data rate and area capacity have been increased up to 1000 times compared to the existing 4G LTE network and the latency has been decreased to 1 ms [1, 2]. In order to effectively provide ultrareliable and low latency communication (URLLC) in the 5G network, a multiaccess edge computing (MEC) server, installed at the base station, becomes a very crucial technology to handle user's requests in real time. Until now, users still need to reach a central server far from the starting point to receive a particular service, which increases latency. However, URLLC can be a reality once MEC provides a service directly next to the base station [3–5].

As wireless network technology develops, more research is being done on how to effectively utilize the Cloud platform. In addition, computation offloading technology has been developed in which user equipment (UE) connects to other platforms to utilize resources because complex tasks could not be performed in the device due to resource problems such as low computing power and weak energy [6].

While mobile phones are the only ones that are mostly connected to the network in the LTE network, in the 5G era various types of terminals such as vehicles, medical devices, robots, drones, and various sensors will always be connected. The information collected from the connected heterogeneous terminals can be used to analyze complex systems, and social internet of things (SIoT) becomes a reality which have not been possible in the LTE network before. However, verification is the first priority or at least a required process for these systems to be implemented before coming into real life. Since the dynamic interactions of complex systems cannot be formalized, the only method to analyze them is a simulation technique. Simulation can be effectively used to analyze complex systems, optimization problems, etc. [7–9].

Taking into account the simulation situation within the MEC platform, the installation of the simulation tool for each MEC application may result in nonefficiency, such as storage duplication and additional overhead which increases when developing the application. This paper proposed the environment since no research has previously been conducted on the MEC platform to provide simulations with a common feature.

2. Background

2.1. Multiaccess Edge Computing. Multiaccess edge computing (MEC) is a crucial method to reduce latency in 5G networks [3–5]. In a 4G network, the terminals had to go to the central server to get information, which makes the delay time longer. In addition, when traffic is concentrated in the central server, congestions, network bottlenecks, and other problems may have occurred which cause network delay. As shown in Figure 1, in the MEC environment, the server is placed very next to the base station, called the edge, so the delay time can be minimized because the terminal can receive a quick response as soon as the request is made.

The European Telecommunications Standards Institute (ETSI) established a standard related to MEC and specified the essential elements [10]. The MEC framework is divided into three levels: the MEC system level, the MEC host level, and the network level. The types of messages exchanged between the components and the overall system structure are determined.

2.2. Docker and Kubernetes

2.2.1. Docker. A Docker is a container-based tool that virtualizes programs. A container is a kind of virtualization that allows a process to run in isolated space [11, 12]. Conventional virtualization used in virtual machines (VMs) was operating system (OS) virtualization; however, OS virtualization's performance is substandard. Therefore, the Docker container virtualized the CPU and dramatically increased performance. The VM method using OS virtualization requires a guest OS per VM, and the overhead greatly increases when the guest OS is used. The Docker runs the container in the form of a process in the Docker engine without a guest OS, so it is much faster because there is no additional overhead.

Because a container operates like an isolated process, it efficiently allocates CPU and system resources even if multiple containers are executed at the same time. By applying Docker's advantages, a research was conducted using Docker in an edge computing platform [13].

2.2.2. Kubernetes. The Docker container has emerged as a breakthrough technology, but since it operates as an individual process, there was no tool to manage it, which makes it difficult to utilize the Docker container efficiently. Consequently, by the necessity of an orchestrator to manage the containers effectively, Google developed Kubernetes (k8s) as an open source platform [14, 15]. Kubernetes connects multiple physical machines (nodes) to form a Cloud, and places the containers on the optimal node, considering the resource and current state of each node. The conceptual structure of Kubernetes is shown in Figure 2. The master node consists of an etcd, scheduler, controller, and API server. The etcd stores information for the Kubernetes system. The scheduler determines a node for a newly created Pod considering the state of the CPU, the memory, and the number of Pods running on nodes, etc. The controller maintains the container to keep the state as intended. The API server is a central communication server that all Kubernetes components must communicate through. The worker nodes are connected to the master node, and the worker nodes run containers. Worker nodes send and receive commands through the API server in the master node.

In Kubernetes, the Pod is a minimum unit which carries several Docker containers. Pods are executed in the worker node as mentioned earlier. A Kubernetes component called "Job" ensures that a task is successfully done by creating one or more Pods to perform certain tasks. The Job is used for the proposed environment described in Section 4.2.1.

A Kubernetes API (kubectl) can be used to generate Kubernetes components such as Pods and Jobs. By default, the API server is accessible only in the master node. In order to access API server in the worker node, a ServiceAccount must be created and the access rules must be defined in a Role component. Afterward, a connection of the Role and ServiceAccount with RoleBinding is required. Finally, when a Pod is created, the authorization to access the API server can be obtained by connecting the defined ServiceAccount to the Pod. In this research, specific tasks are performed by using the Kubernetes API in Pods and Jobs belonging to the worker node. Details of this process are described in Section 4.2.1.

3. Related Works

3.1. Simulation as a Service. Generally, the types of Cloud services are divided into infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS). Each provides a physical resource, a platform to help develop software, and software to users as a service. The advantage of Cloud services is that users can get services only when needed, without having to deploy extra systems. Besides conventional Cloud services, the concept of providing modeling and simulation (M&S) in the form of service has emerged. Since modeling and simulation are separate concepts, they are divided into modeling as a service (MaaS) and simulation as a service (SIMaaS).

There is a study that constructs MaaS and SIMaaS using a traditional Cloud platform [16]. In this study, basic components of the Cloud system (IaaS, PaaS, and SaaS) were constructed using physical infrastructure, and it defined modeling and simulation as a service (MSaaS) using PaaS and SaaS. MSaaS includes modeling as a service, model as a service, and simulation as a service, and these services are provided to the user. Figure 3 shows this in more detail. The model as a service is "a concept of being able to invoke reusable, fine-grained software components across a network" [17]. The modeling as a service is "delivering modeling functionality as a service" [18]. Thus, model as a service becomes some kind of results of modeling as a service.

While this study focused on the architecture for delivering MSaaS from a Cloud perspective, the proposed

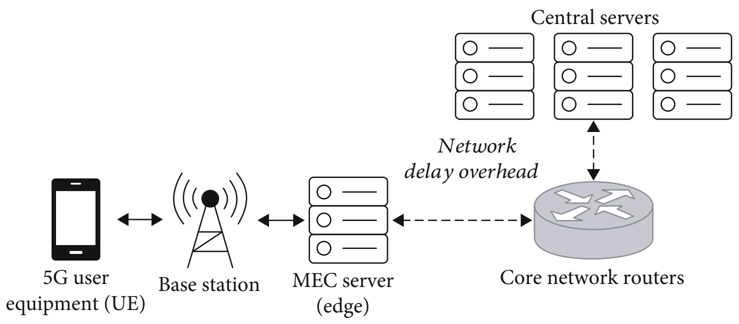


Figure 1: Concept of multiaccess edge computing.

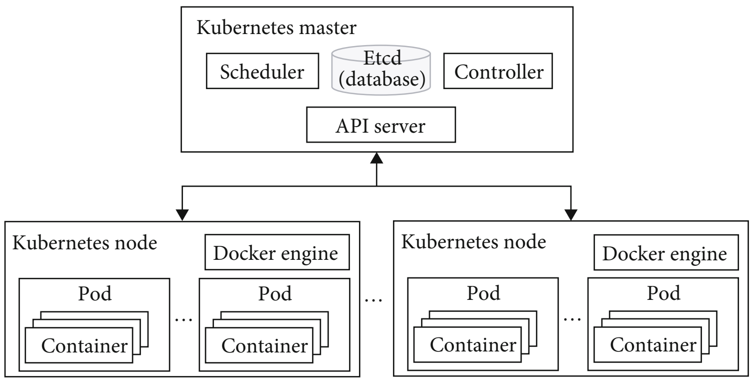


Figure 2: Concept of Kubernetes.

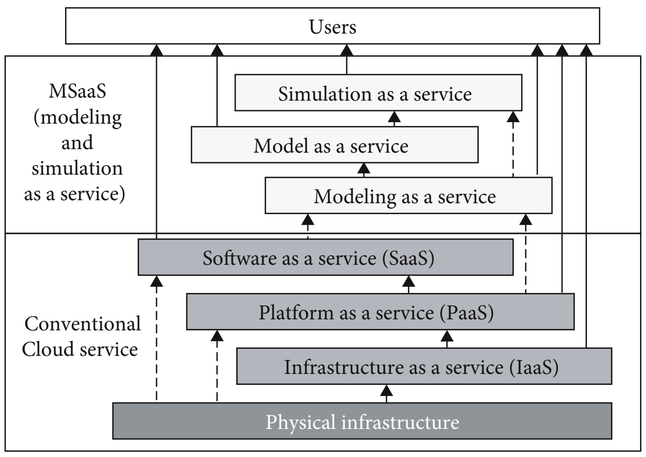

Users
MSaaS (modeling and simulation as a service)
Simulation as a service
Model as a service
Modeling as a service
Software as a service (SaaS)
Conventional Cloud service
Platform as a service (PaaS)
Infrastructure as a service (IaaS)
Physical infrastructure

Figure 3: Relation between conventional Cloud service and MSaaS [16].

research focuses on how to effectively deliver SIMaaS in a MEC environment.

3.2. Simulation in Cloud Platform. There is research on performing simulation using the Amazon Cloud platform [19]. It constructed a system consisting of a master node and worker nodes. The master node manages the simulation and the worker nodes perform the simulation using the resources of the Cloud. The experiments are performed iteratively because the research considered probabilistic variables. The results can be obtained much faster than simulating only in the local machine because the resources of the Cloud can be used to perform experiments on many machines.

The concept of performing a rapid simulation using the Cloud resource of the referenced paper is similar to this research. While the referenced paper focuses only on the simulation, this research considers not only the step of constructing the modeling but also simulating the relationships of heterogeneous models, not just mathematical calculations. In addition, SIMaaS is a kind of an API that enables components of the MEC platform to use the service as a function call. The details of the proposed environment are described in Section 4.

4. MEC-Based Simulation as a Service

4.1. MEC Platform Structure. As a mobile communication infrastructure for simulation as a service, we are developing a MEC platform that conforms to the reference architecture [20] and specifications presented by ETSI. The MEC reference architecture provides the service API that the MEC platform provides for MEC apps as summarized in Table 1.

In addition to ETSI-based standard MEC services and APIs, we developed a digital twin service which manages virtual world software replications of real-time status 5G terminals in the real world [21]. The proposed MEC platform includes a digital twin service, and APIs are implemented using the Kubernetes Pod component. Based on the information and location of terminals within the 5G network, the

Table 1: A set of APIs for MEC apps.

API and Std. number	Description
Application enablement API (ETSI GS MEC 011)	Service-related functionality includes registration, discovery, and event notifications between MEC apps and MEC platform
Radio network information service API (ETSI GS MEC 012)	Define how the current wireless network status information is obtained so that the MEC application or MEC platform can optimize the service according to the state of the wireless network
Location service API (ETSI GS MEC 013)	Define how the location information of the terminal is obtained so that the MEC application or the MEC platform can provide services based on the location of the terminal being serviced
UE identity service API (ETSI GS MEC 014)	Define how the MEC application or MEC platform can register a tag (ID) for a specific terminal in service, enabling the carrier to set rules (ex., traffic) for each device
Bandwidth management service API (ETSI GS MEC 015)	Define how MEC applications and the MEC platform update or receive the required bandwidth information
UE app API (ETSI GS MEC 016)	Define the lifecycle management between UE application (client application) in the UE and the user application lifecycle management proxy in the MEC platform

digital twin service creates digital twin models that correspond to managed terminals and synchronizes them in real time by reflecting data collected from them. Information and location of the terminal are obtained through the location service API and UE identity service API. The digital twin service not only provides MEC applications with real-time status information on terminals but also provides simulation capability to predict the future behavior of terminals according to the state changes in the 5G network. The state of the 5G network is obtained through radio information service API and bandwidth management service API. Figure 4 shows the structure of the MEC platform in which the digital twin service operates.

The digital twin service includes the modeling service and simulation service. This paper deals with simulation service, and it is described in more detail in Section 4.2.

4.2. Development of Simulation as a Service

4.2.1. Structure of Simulation as a Service. In this paper, we defined simulation as a service (SIMaaS) which performs a simulation and returns the simulation result depending on the request of a MEC application in a Kubernetes-based MEC platform. The structure of the environment to provide SIMaaS is shown in Figure 5.

A digital twin service includes a modeling service and a simulation service. When a simulation is requested from the MEC application to the digital twin service, the modeling service identifies the model information suitable for the requested situation, creates the schema information of the simulation model in XML format, and passes it to the simulation service. Currently, the modeling service is not implemented. We designed the whole structure for the integrity of the entire environment, so the implementation of the modeling service has remained as future work. In this study, we assume that the modeling service's result is provided to the simulation service as a predefined model.

The XML structure of the model schema for the simulation service is shown in Figure 6. The ModelStructure node defines the name of the message type and port value for the simulation model. The child node of the ModelStructure node is a NModel node which means network model. It represents the connected model of atomic models, which no longer splits to smaller elements and can play independent roles. The component node, which is the child node of the NModel node, defines the atomic model. The attribute node, which is the child of the component node, contains the variable information used to initialize the atomic model, and one component node can have multiple attribution nodes. The coupling node specifies the connection between the input and output ports, which are the paths through which messages are exchanged between the models. It connects the models by linking "FromPort" of "FromModel" to "ToPort" of "ToModel." The port of the network model can also be defined in port node. The port node specifies the name of the in and out port and the port number.

A simulation service Pod is a kind of server that processes requests when it receives simulation requests. The simulation service receives a schema of the simulation model in XML file format and simulation information from the modeling service. After receiving files, the simulation service generates the simulation model source code compatible with the Adevs environment by combining the atomic models as specified in the model structure XML. The simulation service has the source code of the predefined atomic models. Then, the simulation service generates a Makefile and executes the script file which runs the generated Makefile to compile and get the executable file.

When the executable file of the simulation model is generated, the simulation service creates the simulation engine Jobs corresponding to the number of simulations to be performed using the Kubernetes API. Since the simulation service is a Pod belonging to the worker node, it cannot use the Kubernetes API by default. Therefore, it acquires the

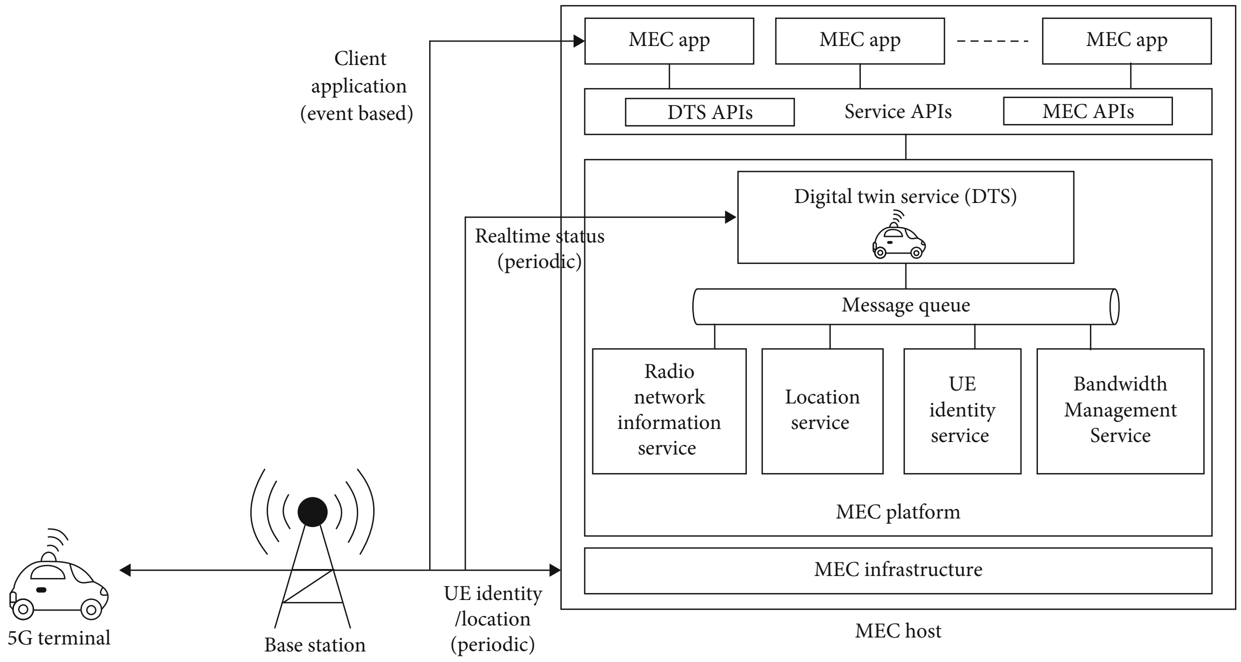


Figure 4: Structure of a digital twin service-based MEC platform.

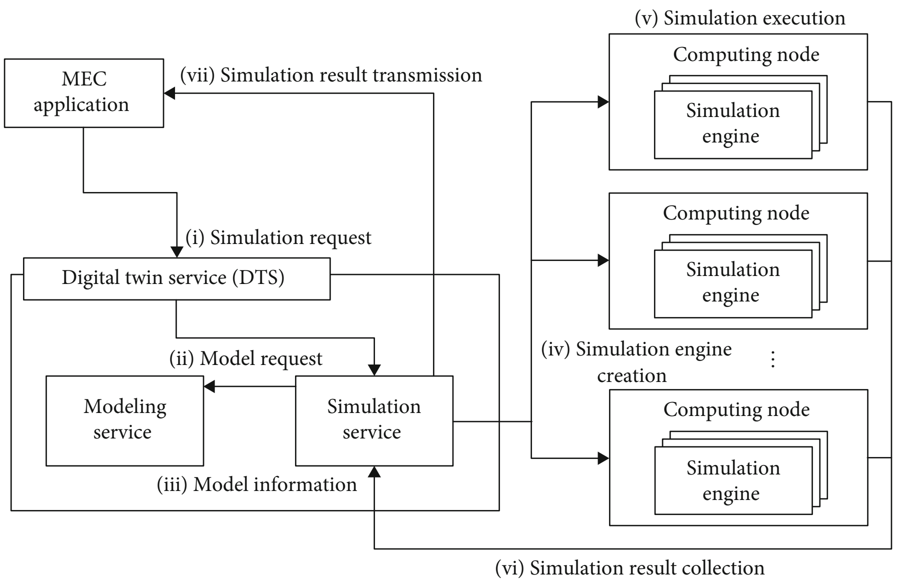


Figure 5: Structure of simulation as a service.

authority to use the Kubernetes API through a series of processes, such as Rolebinding described in Section 2.2.2. A detailed example of a yaml file format application is shown in Figure 7. The yaml file creates a ServiceAccount named simulation-service, which then creates a Role named pod-reader, then connects simulation-service and pod-reader in RoleBinding. Then, when a Pod is created, it finally gets permission to use the Kubernetes API by specifying “serviceAccountName: simulation-service.”

When the simulation engine Job is created, a program that connects the Job to the simulation service is executed automatically. The simulation service and simulation engines exchange data through TCP socket communications. However, due to the characteristic of the Kubernetes Pod, which dynamically assigns an IP address whenever a Pod is executed, the IP address cannot be known when opening a socket. To solve this problem, the simulation engine Job also obtains the authority to use the Kubernetes API, and then uses the API (kubectl-described Pod) to find the IP address of the simulation service Pod. When the simulation engine finds the IP address of the simulation service and establishes the connection, the simulation service sends the executable simulation model file. The simulation engine Job performs a simulation by executing the received file, and when the

```
<ModelStructure MsgType="msgType" PortValue="IO_Type">
    <NModel Name="modelName">

        <Component Name="comp1" Type="Atomic"/>

        <Component Name="comp2" Type="Atomic">
            <Attribute Name="Distribution" Value="5"/>
            <Attribute Name="Type" Value="input"/>
        </Component>

        <Coupling FromModel="comp1" FromPort="out" ToModel="comp2" ToPort="in"/>
        <Coupling FromModel="comp2" FromPort="out" ToModel="comp1" ToPort="in"/>

        <Port>
            <In Name="In" PortNum="1000"/>
            <Out Name="Out" PortNum="2000"/>
        </Port>
    </NModel>
</ModelStructure>
```

Figure 6: XML schema of simulation model.

```
apiVersion: v1                                  ---
kind: ServiceAccount                            kind: RoleBinding
metadata:                                       apiVersion: rbac.authorization.k8s.io/v1beta1
  name: simulation-service                      subjects:
---                                             -kind: ServiceAccount
kind: Role                                        name: simulation-service
apiVersion: rbac.authorization.k8s.io/v1          namespace: default
metadata:                                       roleRef:
  name: pod-reader                                apiGroup: rbac.authorization.k8s.io
  namespace: default                              kind: Role
rules:                                            name: pod-reader
- apiGroups: [""]                               ---
  resources: ["pods"]                           apiVersion: v1
  verbs: ["get", "watch", "list", "create"]     kind: Pod
                                                spec:
                                                  serviceAccountName: simulation-service
```

Figure 7: A method to acquire the authorization of a Kubernetes master.

simulation ends, sends the result to the simulation service. Since the simulation engine does not need to do further tasks after the simulation, it is created as a Kubernetes Job for efficient memory utilization of the entire server. The simulation service collects the simulation results received from the simulation engines and sends them to the MEC application which requested the simulation.

The Kubernetes system does not delete the data even though the Job is finished. Therefore, when all simulations are finished, the simulation service automatically deletes created Jobs using the kubectl command.

4.2.2. Simulation Engine. The expected situations for simulating via the MEC platform are mostly constructed with a discrete event system. A discrete event system is an "event-driven system in which discrete states are updated depending entirely on the occurrence of asynchronous discrete events over time" [22].

Generally, objects that make individual decisions are connected in the MEC network. Each decision becomes an event to affect another objects' behavior. This is called the "event-driven system" which I mentioned before. A discrete event system specification (DEVS) formalism [7] describes the discrete event system.

The model representation is based on DEVS formalism, so a DEVS-based simulation engine is required. Several DEVS-based simulation engines are described in Table 2. Among these simulation engines, we decided to use Adevs [23] as this environment's simulation engine. Because SIMaaS is used in a real-time system, the execution speed is an important factor.

According to the benchmark results of research conducted by Van Tendeloo and Vangheluwe [24], Adevs has the least CPU time used, which means that Adevs is the fastest among DEVS-based simulation engines. The benchmark ranking is noticed in Table 2.

5. Case Study

5.1. Scenario Description. The problem to be solved in the case study is to find out the optimal gate in a situation where autonomous vehicles (AVs) and manually driven vehicles

Table 2: Characteristics of DEVS-based simulation engines [24].

DEVS-based simulation engines	Characteristics	Benchmark ranking
Adevs	(i) Lightweight C++ library (ii) Fast speed (iii) Recompiling required after every model edit (iv) Free license	1
PowerDEVS	(i) C++-based engine (ii) Graphical modeling environment (iii) GPL license	2
Python PDEVS	(i) Python-based engine (ii) Fast prototyping models (iii) Apache license	3
CD++	(i) C++-based engine (ii) Graphical modeling environment (iii) License unspecified	4
DEVS-Suite	(i) Java-based engine (ii) Graphical modeling environment (iii) LGPL license	5

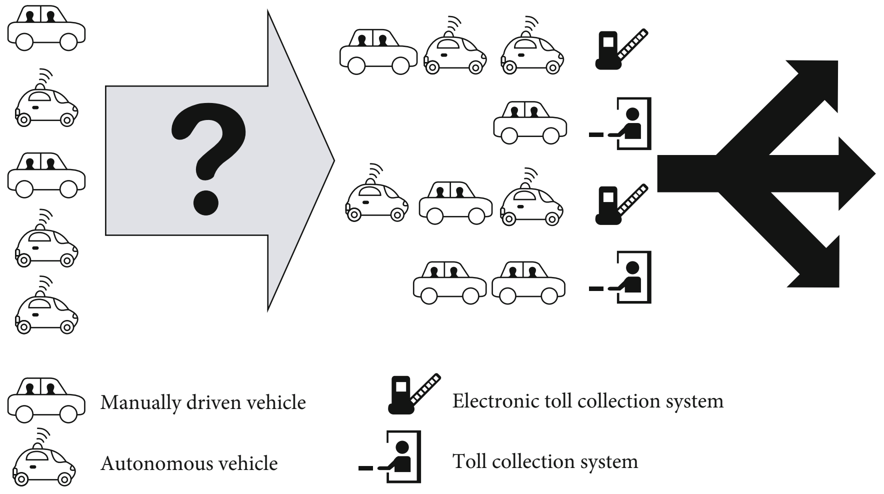


Figure 8: Tollgate selection problem on a highway.

(MVs) are mixed and are made to pass through a highway toll collection system (TCS; as known as tollgate). The abstract situation is described in Figure 8 below.

When a vehicle approaches the tollgate, it is connected to the regional MEC server network. Then, a decision-making app (MEC client app) in the vehicle requests the optimal gate to the tollgate selection MEC app running in the MEC server. The MEC app calls the simulation service to acquire the average processing time and the average queue waiting time of each gate based on the current situation. The data are required to decide what is the optimal gate. In the case study, the modeling service is not implemented, so we assumed a certain situation, which is described in Section 5.2. Consequently, when the MEC app requests simulation to the simulation service, it sends the XML schema of the simulation model. The simulation service performs simulations based on the MEC app's request using multiple computing nodes. After finishing the simulation, the simulation service sends the simulation results to the MEC app. Then, the MEC app analyses the simulation result and gets the optimal gate. Finally, the MEC app sends the result of the simulation to the MEC client app. The process is described in Figure 9.

The arrival time of the vehicles and the processing time of a tollgate are stochastic, thus a Monte-Carlo simulation [25], which is a simulation method to solve the probabilistic problems by simulating repeatedly, is required. Because the proposed SIMaaS utilizes Cloud infrastructure to perform multiple simulations, it can be a decent option for the case study.

5.2. Simulation Model Description. The structure of the model for this case study is shown in Figure 10. The generator model creates vehicle objects periodically following a certain distribution and sends them to the buffer model. When a vehicle object is created, the heading direction, car type, and pay type are randomly assigned. The heading direction is the

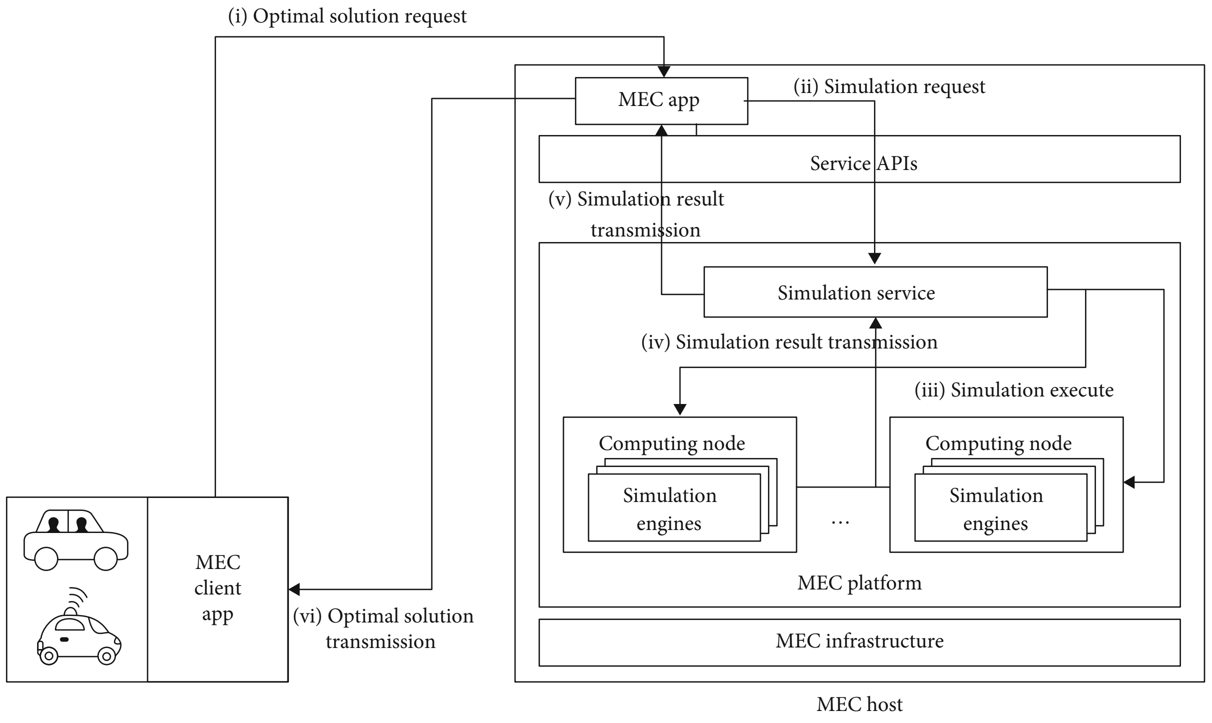


Figure 9: Process of simulation service request handling.

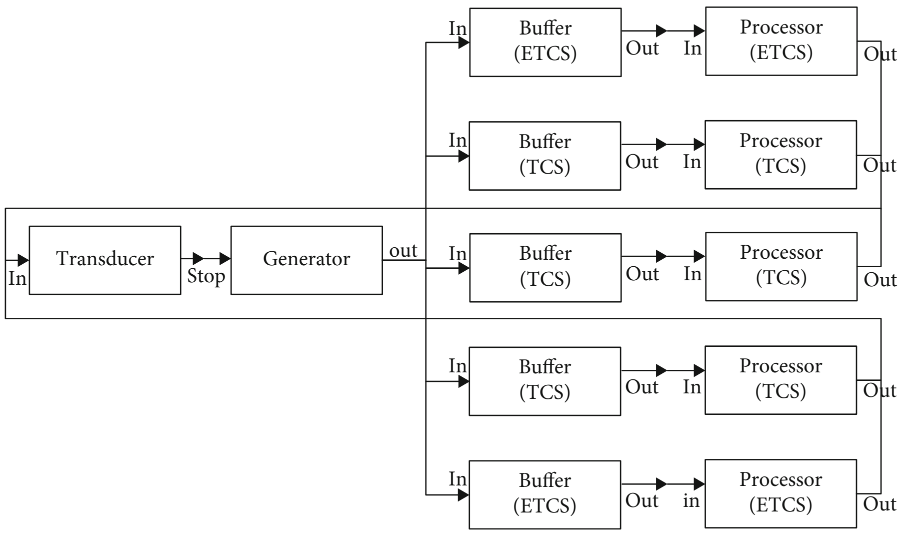


Figure 10: Structural model of toll collection systems.

heading way out of the tollgate, and it can have the value of right, straight, and left. The car type is a vehicle's driver type. It can be autonomous or manual. The pay type is a payment method of a vehicle. It can have the value of electronic automatic or manual.

The buffer model transmits the stored vehicle objects with the first in first out (FIFO) manner to the connected processor model. The processor model is the tollgate, and it handles the vehicle objects. Its types are an electronic toll collection system (ETCS) or a toll collection system (TCS). ETCS charges the fare automatically without stopping vehicles. In TCS, the fare is manually charged by a human. Each type of processor has a different processing time because the characteristics of the types differ. Generally, the processing time of the ETCS model is faster than the TCS model. For this case study, two ETCS processors and three TCS processors are used for constructing the tollgate models as shown in Figure 10. The ETCS processors are located top and bottom, while the TCS processors are located in the middle.

The transducer model collects the finished objects passed by the processors to analyze the simulation results. It calculates average waiting time, average service time, average queue waiting time, average queue length, average electric queue length, and average manual queue length. When the object count reaches the end of the simulation count, the transducer produces a final simulation result and sends a stop command to the generator model.

We predefined the XML format simulation of Figure 10 on behalf of the modeling service. When the simulation service receives the XML model, it automatically generates an executable file as described in Section 4.2.1, then creates simulation engines to run simulation. After the end of the simulations, the simulation service collects the results from simulation engines.

When a vehicle model is created, it is probabilistically determined whether it is an autonomous vehicle or a manually driven vehicle. In the case of an autonomous vehicle, it is assumed that the payment method is only electronic. The manually driven vehicle can have two payment methods, electronic and manual, which are also probabilistically determined. ETCS is available only if the payment method is electronic, and TCS can be used regardless of the payment method if the type of vehicle is manual. The situation in which AV uses TCS is not considered because this case study assumes a fully autonomous driving situation.

When the generator model creates vehicle objects to send to the buffer model, the optimal buffer is determined by Algorithm 1. Since one buffer is connected to each processor, sending to a particular buffer means sending to the connected processor. Algorithm 1 has D, T_c, T_p, T_q, B, and P as parameters. The components of the parameters mean heading direction after passing through the TCS, car type, pay type, queue type, buffer size, and self-determination probability, respectively.

T_c can have AUTONOMOUS or MANUAL as a value, and T_p and T_q can have ELECTRONIC or MANUAL as a value. The self-determination probability (P) is the probability that the driver of the manually driven vehicle arbitrarily makes a different decision without following the calculated destination by the MEC application. Since rand ()%10 has a range of 0 to 9, P also has a range of 0 to 9, and when P is 0, the probability of making another decision is 0%, and when it is 9, the probability goes 100%. Algorithm 1 is connected to Algorithm 2 and finally finds the optimal buffer. In the INITIALIZATION phase, it calculates the searching range (start, end) for Algorithm 2. In the BUFFER DECISION step, the final result is obtained by calling Algorithm 2 with T_c, T_p, T_q, B, *start*, *end*, and C as a parameter. C is a parameter for self-determination probability (P) calculation and it has a value of 0 or 1. When considering self-determining situation, c is 1. Otherwise, c is 0. In the case of C equals 1, if both the car type is MANUAL and the probability of P is satisfied, add an extra task to select a buffer other than the previously selected buffer. The final result is achieved by calling Algorithm 2 with C set to 1.

Algorithm 2 finds the buffer matching the payment type of the vehicle within the range of (*start*, *end*) and selects the smallest size of buffer as a result. If C equals 1, the MANUAL car type is solely considered. When the manually driven vehicle's payment method is AUTOMATIC, it is basically allocated to the automatic buffer. Since the manually driven vehicle's driver can pay directly, Algorithm 2 considers both types of the buffer and recalculates the final result by choosing the smallest size of the buffer.

5.3. Model Validation and Verification. The toll collection system model is constructed with a basic generator, a buffer, and processor (GBP) models. To know if the model follows the real system and is completely implemented, we calculated the ideal result using the queueing theory and compared the result. The assumed system consists of one generator, one buffer, and one processor. The testing model for validation is described in Figure 11, and it is a typical $M/M/1$ queue model with an arrival rate (λ) that follows a Poisson process and a service rate (μ) that follows an exponential distribution with one processor. For this experiment, λ and μ are set to 10 and 12 each, and the unit time is set to seconds.

The equations of the $M/M/1$ queue are described as follows: T is the average amount of time that a customer spends in the system (1), W is the average amount of time spent in a queue (2), and N_Q is the average number of customers in the buffer (3). According to the queueing theory, when λ equals 10 and μ equals 12, T is 0.5, W is 0.42, and N_Q is 4.17.

$$T = \frac{1}{\mu - \lambda} = \frac{1}{12 - 10} = 0.5, \tag{1}$$

$$W = \frac{1}{\mu - \lambda} - \frac{1}{\mu} = \frac{1}{12 - 10} - \frac{1}{12} = 0.42, \tag{2}$$

$$N_Q = \frac{\lambda}{\mu - \lambda} - \frac{\lambda}{\mu} = \frac{10}{12 - 10} - \frac{10}{12} = 4.17. \tag{3}$$

To verify the implementation, an experiment is conducted simulating 1,000,000 processing objects for 10 times and calculating the average value. The result is as follows: T is 0.53, W is 0.44, and N_Q is 4.3. It is similar to the queueing theory's calculation result; therefore, the implementation is verified.

When it comes to validating the scenario described in Section 5.1, it is hard to obtain real-world data at this point. Because fully activated autonomous vehicles are not commercialized, there are no comparable real-world data assumed in this case study. Nevertheless, the model follows the queueing theory; thus, if we have real data, the model will act like a real situation.

5.4. Experiment Design and Results. The arrival rate (λ) of a generator assumed a crowded time of following the Poisson distribution with 3600 units per hour. We assumed that the service rate (μ) of the processors follows exponential distribution of 1200 units per hour for ETCS and 257 units per hour for TCS. To sum up, $\lambda = 3600$, $\mu(\text{ETCS}) = 1200$, and $\mu(\text{TCS}) = 257$.

AV solely enters the tollgate specified by the MEC server, while MV can enter another tollgate arbitrarily without following the decision made by the MEC server. The arbitrary probability is assumed as 20% (P equals 2) for this case study. This is because people often ignore GPS navigation

```
Find the optimal buffer
Parameters: $D$(direction), $T_c$(car type), $T_p$(pay type), $T_q$ (queue type), $B$(buffer size), $P$(self-determination probability).
  INITIALIZATION:
1:  If D = = LEFT then
2:      set start ⟵ 0
3:      set end ⟵ B/3 − 1
4:  End if
5:  Else if D = = STRAIGHT then
6:      set start ⟵ B/3
7:      set end ⟵ start + B/3 − 1 + B%3
8:  End if
9:  Else then
10:     set start ⟵ end + 1 − B/3
11:     set end ⟵ B − 1
12:     If B/3 = = 0 then
13:         set start ⟵ start − 1
14:     End if
15: End else
16: If start < 0 then
17:     set start ⟵ 0
18: End if
BUFFER DECISION:
19: set result ⟵ Algorithm 2($T_c$, $T_p$, $T_q$, $B$, $start$, $end$, 0)
20: If $T_c$ = = MANUAL then
21:   If P < rand ()%10 then
22:     set result⟵Algorithm 2($T_c$, $T_p$, $T_q$, $B$, $start$, $end$, 1)
23:     End if
24: End if
FINALIZATION:
25: return result
```

Algorithm 1

```
Find the optimal buffer
Parameters: $T_c$(car type), $T_p$(pay type), $T_q$(queue type), $B$(buffer size), $S$(start), $E$(end), $C$(self-determination condition).
  INITIALIZATION:
1:  set result⟵−1
2:  set min ⟵ ∞
  BUFFER DECISION:
3:  do
      LOOP:
4:      for i = S to E do
5:          If C = = 1 && $T_p$ = = ELECTRONIC
            ||$T_q$ = = $T_p$ then
6:              If B < min
7:                  set min ⟵ B
8:                  set result ⟵ i
9:              End if
10:         End if
11:     End for
12:     set S ⟵ S − 2
13:     set E ⟵ E + 2
14:     If S < 0 then
15:         set S ⟵ 0
16:     End if
17:     If E > B − 1 then
18:         set E ⟵ B − 1
19:     End if
20: while result = = −1
  FINALIZATION:
21: return result
```

Algorithm 2

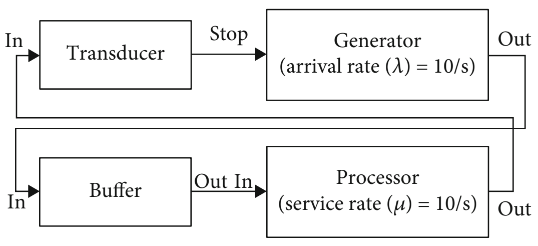


Figure 11: Queueing theory validation model.

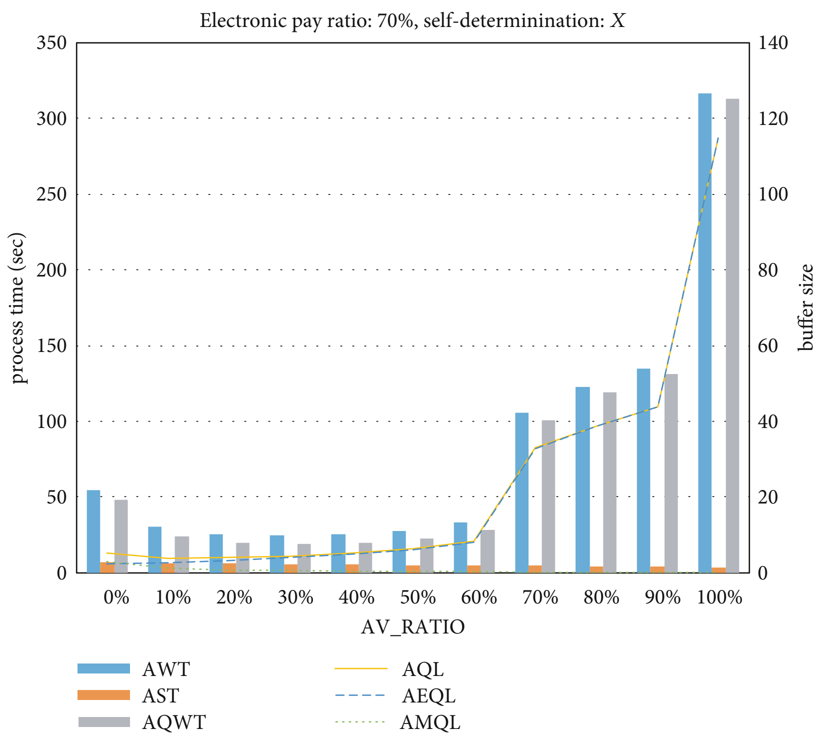
information and make different decisions based on personal judgment.

Experiments were conducted by changing the ratio of AV from 0% to 100%. There is a study on analyzing the driving stress of MV drivers in circumstances where AV and MV are operated together [26]. In a similar context, this experiment is aimed at analyzing the delay in passing tollgates depending on the ratio of AV.

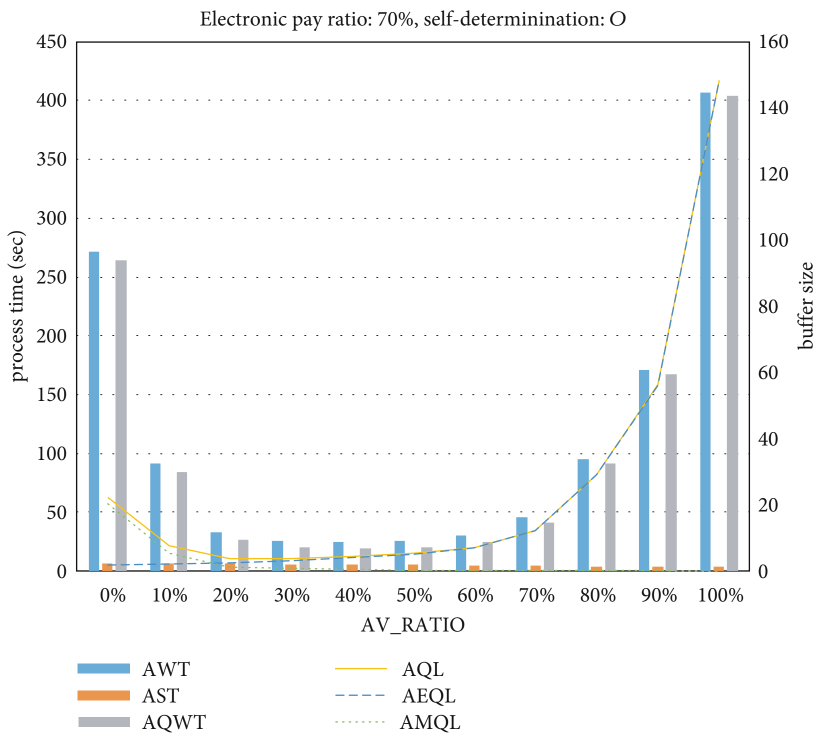
Because all situations (generating vehicles, assigning to a specific buffer, and processing time in a processor) are probabilistic, a Monte-Carlo simulation is applied to produce results through repeated executions. In each case, we simulated 100 times for 10,000 vehicle passes, and took an average. The experiment results are shown in Figure 12.

The information collected in the experiment are average waiting time (AWT), average service time (AST), average queue waiting time (AQWT), average queue length (AQL), average electric queue length (AEQL), and average manual queue length (AMQL). The AWT is the sum of the time it takes for objects to be created and passed through processors divided by the number of objects. The AST is the sum of all the time taken by the processors divided by the number of objects. The AQL is the sum of the queue lengths at the time the object enters the processor divided by the number of objects. Similar to AQL, AEQL is a summation of the ETCS buffer, and AMQL is a summation of the TCS buffer.

From the experiment results, we can figure out the tollgate delay tendency based on various circumstances. In the case where a self-deterministic mode is available, AST and AQL increase. Comparing Figure 12(a) and Figure 12(b), Figure 12(b) represents a circumstance where the self-deterministic mode is on, and it's AST is bigger than that of Figure 12(a). Also, if the ratio of using ETCS is high, AST tends to be smaller. Comparing Figure 12(a) and Figure 12(e), Figure 12(a)'s ETCS-using ratio is 70%, and Figure 12(e)'s ETCS-using ratio is 50%. The former's AST is

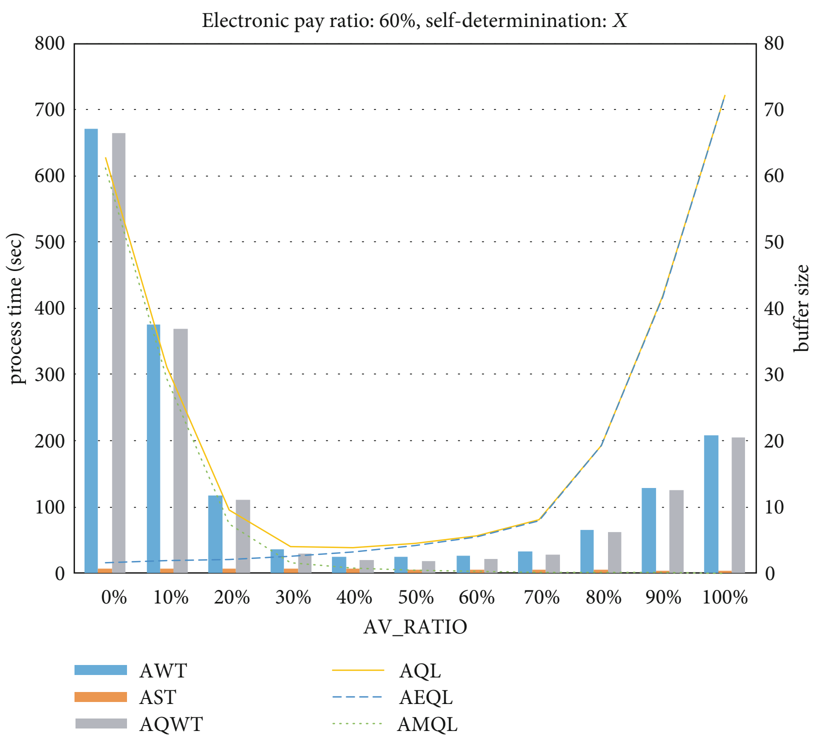


(a)

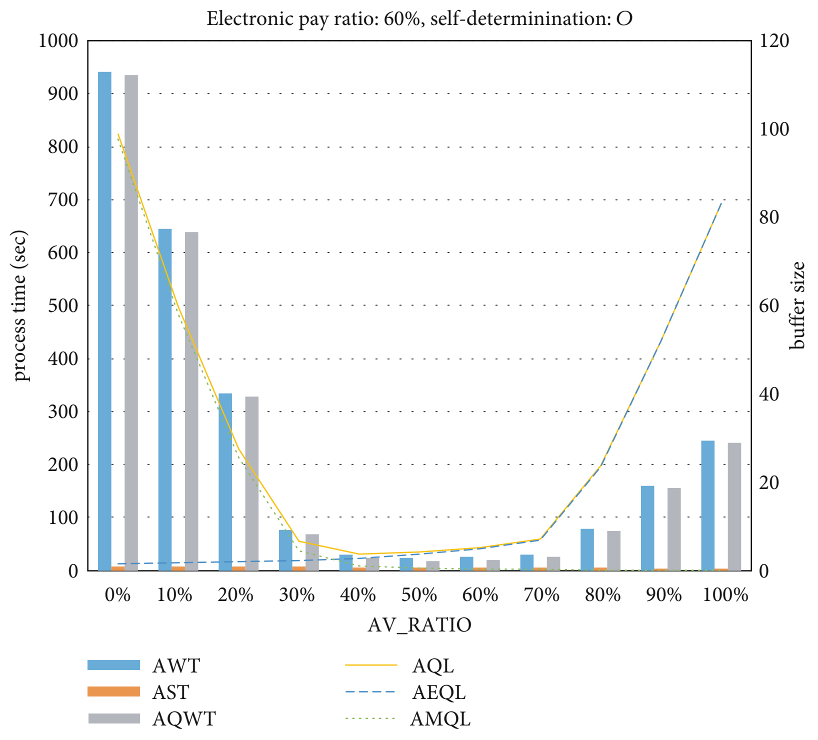


(b)

Figure 12: Continued.

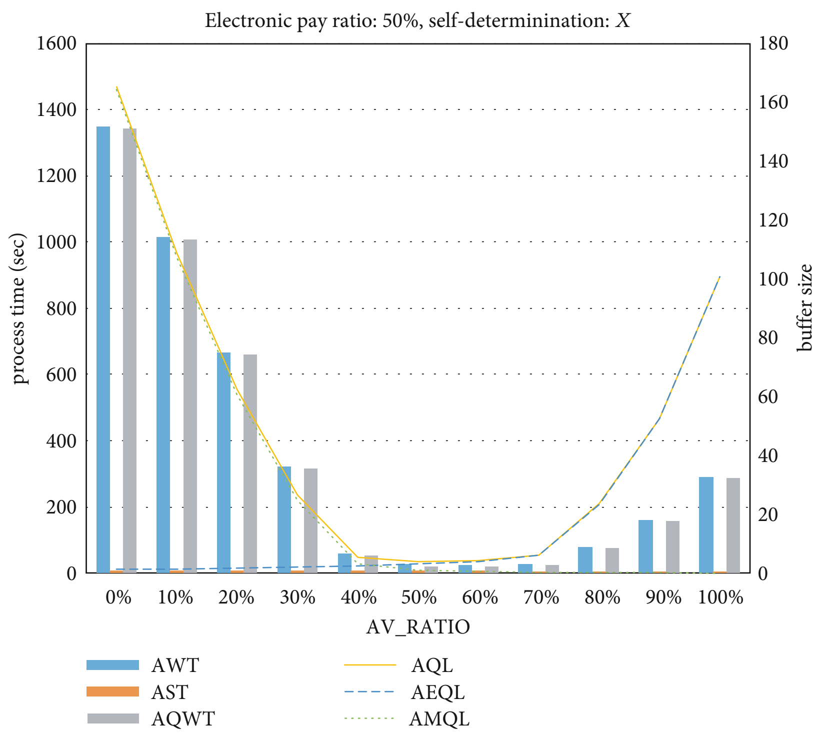


(c)

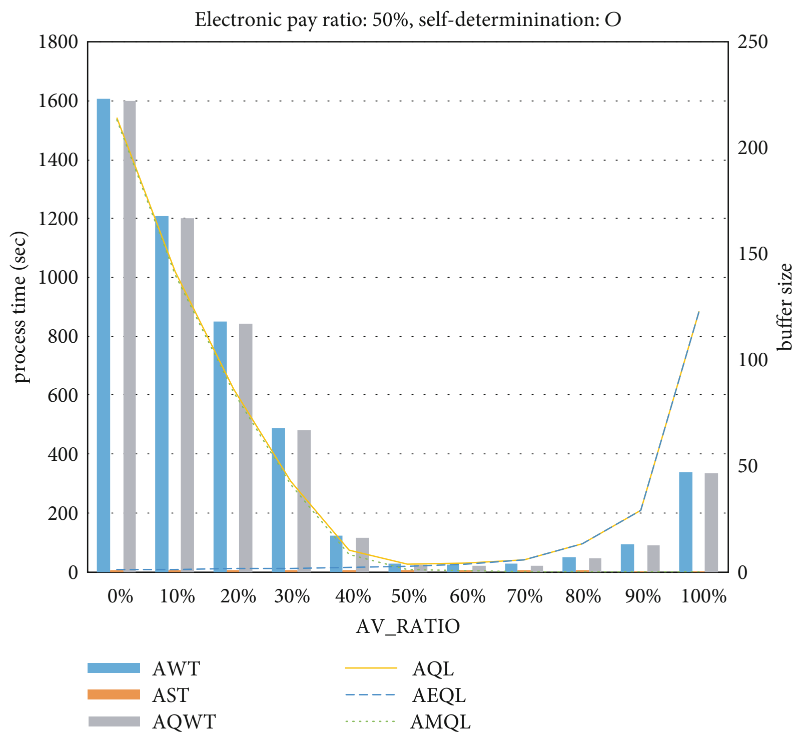


(d)

Figure 12: Continued.



(e)



(f)

Figure 12: Experiment results of the case study.

significantly smaller than the latter. In every case, AST and AQL increase when the AV_RATIO is too small or too big.

This experiment shows an example of the result when vehicles in various circumstances request optimal gates. Therefore, when vehicles approach MEC-installed tollgates and request the optimal gate information, the MEC app immediately transmits the gate information in any circumstances by using the SIMaaS.

6. Conclusion

In this paper, we explained a Kubernetes-based multiaccess edge computing platform to provide ultrareliable and low latency communication service to user equipment connecting to the 5G network. Also, we designed and implemented SIMaaS that provides the simulation for the MEC apps to run in the MEC platform. SIMaaS performs multiple simulations using numerous simulation engines spread in many computing nodes. In the case study, we constructed a highway toll collection system where autonomous and manually driven vehicles are passing through. When a vehicle approaches the tollgate, it requests the optimal gate information to the MEC server. Monte-Carlo simulation is carried out to handle probabilistic situations such as vehicles' arrival rate and tollgates' service rate. Since MEC apps do not solely solve to find out the optimal path to the tollgates for vehicles, SIMaaS is called and transmits the simulation results back to the MEC apps. After analyzing the results, the MEC apps notifies optimal tollgate information to the MEC client, which is a vehicle.

The experimental results show that the processing time is reduced as the ratio of using an electronic toll collection system increases and the ratio of drivers' self-deterministic behavior decreases.

Moreover, besides this case study, in the near future, there will be thousands of objects connecting over the 5G network. To analyze relations of the complex intertwined heterogeneous objects, the proposed SIMaaS can be a good option.

For future work, we are planning to implement a modeling service which analyzes the objects connected to the MEC network and makes a digital twin model of objects. With the modeling service, the dynamically changing system can be simulated without manual model construction.

Data Availability

The data availability statements are listed as follows: (1) The experiment results data used to support the findings of this study are available from the corresponding author upon request. (2) The software code data used to support the findings of this study have not been made available because the data belonged to my institute.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by "The Cross-Ministry Giga KOREA Project" grant funded by the Korea government (MSIT) (No. GK19P0400, Development of Mobile Edge Computing Platform Technology for URLLC Services).

References

[1] J. G. Andrews, S. Buzzi, W. Choi et al., "What will 5G be?," *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 6, pp. 1065–1082, 2014.

[2] N. Panwar, S. Sharma, and A. K. Singh, "A survey on 5G: the next generation of mobile communication," *Physical Communication*, vol. 18, pp. 64–84, 2016.

[3] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: a survey of the emerging 5G network edge Cloud architecture and orchestration," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1657–1681, 2017.

[4] C.-H. Hsu, S. Wang, Y. Zhang, and A. Kobusinska, "Mobile edge computing," *Wireless Communications and Mobile Computing*, vol. 2018, Article ID 7291954, 3 pages, 2018.

[5] P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila, and T. Taleb, "Survey on multi-access edge computing for internet of things realization," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 2961–2991, 2018.

[6] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge Cloud computing," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, 2015.

[7] B. P. Zeigler, T. G. Kim, and H. Praehofer, *Theory of Modeling and Simulation*, Academic press, 2000.

[8] A. M. Law, W. D. Kelton, and W. D. Kelton, *Simulation Modeling and Analysis*, McGraw-Hill, 2000.

[9] K. H. Lee, J. H. Hong, and T. G. Kim, "System of systems approach to formal modeling of CPS for simulation-based analysis," *ETRI Journal*, vol. 37, no. 1, pp. 175–185, 2015.

[10] F. Giust, X. Costa-Perez, and A. Reznik, "Multi-access edge computing: an overview of ETSI MEC ISG," *IEEE 5G Tech Focus*, vol. 1, no. 4, 2017.

[11] J. Turnbull, *The Docker Book: Containerization is the New Virtualization*, Lulu.com, 2014.

[12] C. Anderson, "Docker (software engineering)," *IEEE Software*, vol. 32, no. 3, pp. 102–1c3, 2015.

[13] B. I. Ismail, E. M. Goortani, M. B. A. Karim et al., "Evaluation of Docker as edge computing platform," in *2015 IEEE Conference on Open Systems (ICOS)*, pp. 130–135, Bandar Melaka, Malaysia, 2015.

[14] K. Hightower, B. Burns, and J. Beda, *Kubernetes: Up and Running: Dive into the Future of Infrastructure*, O'Reilly Media, Inc., 2017.

[15] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Queue*, vol. 14, no. 1, pp. 70–93, 2016.

[16] E. Cayirci, "Modeling and simulation as a Cloud service: a survey," in *2013 Winter Simulations Conference (WSC)*, pp. 389–400, Washington, DC, USA, 2013.

[17] D. Roman, S. Schade, A. J. Berre, N. R. Bodsberg, and J. Langlois, *Model as a service (MaaS)*, AGILE Workshop: Grid Technologies for Geospatial Applications, Hannover, Germany, 2009.

[18] S. Popoola, J. Carver, and J. Gray, *Modeling as a Service: a Survey of Existing Tools*, MODELS (Satellite Events), 2017.

[19] H. Wang, Y. Ma, G. Pratx, and L. Xing, "Toward real-time Monte Carlo simulation using a commercial Cloud computing infrastructure," *Physics in Medicine & Biology*, vol. 56, no. 17, pp. N175–N181, 2011.

[20] ETSI MEC APIs, https://forge.etsi.org/gitlab/mec.

[21] E. Glaessgen and D. Stargel, "The digital twin paradigm for future NASA and US Air Force vehicles," in *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference 20th AIAA/ASME/AHS Adaptive Structures Conference 14th AIAA*, pp. 1–14, Honolulu, Hawaii, 2012.

[22] S. Choi, K.-M. Seo, and T. Kim, "Accelerated simulation of discrete event dynamic systems via a multi-fidelity modeling framework," *Applied Sciences*, vol. 7, no. 10, p. 1056, 2017.

[23] J. Nutaro, "Adevs (a discrete event system simulator)," in *Arizona Center for Integrative Modeling & Simulation (ACIMS)*University of Arizona, Tucson1999, http://www.ece.arizona.edu/nutaro/index.php.

[24] Y. Van Tendeloo and H. Vangheluwe, "An evaluation of DEVS simulation tools," *Simulation*, vol. 93, no. 2, pp. 103–121, 2016.

[25] R. Y. Rubinstein and D. P. Kroese, *Simulation and the Monte Carlo Method*, Wiley, 2016.

[26] Y. Nishimura, A. Fujita, A. Hiromori et al., "A study on behavior of autonomous vehicles cooperating with manually driven vehicles," in *2019 IEEE International Conference on Pervasive Computing and Communications PerCom*, pp. 212–219, Kyoto, Japan, 2019.