A Dynamic Partitioning Framework for Edge-Assisted Cloud Computing

Zhengjia Cao[1,2], Bowen Xiao[1,2], Haihan Duan[1,2], Lei Yang[3], and Wei Cai[1,2](✉)

[1] The Chinese University of Hong Kong, Shenzhen, Shenzhen, China
{zhengjiacao,bowenxiao,haihanduan}@link.cuhk.edu.cn
caiwei@cuhk.edu.cn

[2] Shenzhen Institute of Artificial Intelligence and Robotics for Society, Shenzhen, China

[3] South China University of Technology, Guangzhou, China
sely@scut.edu.cn

Abstract. Edge computing is commonly adapted to reduce network delay and fluctuation in the traditional cloud-terminal architecture. Many research teams have been dedicated to algorithms and optimizations in the cloud-edge data cache and computation offloading schemes, while there is a blank in practical implementation to facilitate such a paradigm. In this work, we proposed a component-based framework that facilitates dynamic partitioning of a software program. We have designed and implemented the very first test-bed for further optimizing the components distribution strategy among cloud, edge, and terminal devices. Experiments have revealed the characteristics of components execution in the proposed architecture, showing that the system can improve computing performance under the real-world unstable network environments.

Keywords: Cloud computing · Edge computing · Software decomposition · Offloading · Distributed system

1 Introduction

With abundant computational resources as a guarantee, cloud computing is becoming the main direction of software evolution. The software in recent years, such as cloud-hosted augmented reality, online video editing, games, etc., often demands assistance from cloud computing technologies [3] in data storage and computing resources [14]. However, accessing the remote cloud server introduces inevitable network delay. As a supporting infrastructure of cloud computing, edge computing [6,16] utilizes hardware facilities at the edge of the network as

This work was supported by Project 61902333 supported by National Natural Science Foundation of China, by the Key Area R&D Program of Guangdong Province with grant No. 2018B030338001, by Project 61972161 supported by National Natural Science Foundation of China, by the Shenzhen Institute of Artificial Intelligence and Robotics for Society (AIRS).

M. Qiu (Ed.): ICA3PP 2020, LNCS 12453, pp. 215–229, 2020.
https://doi.org/10.1007/978-3-030-60239-0_15

relay nodes for reducing latency, pushing the cloud services closer to the terminal end to avoid backbone network communication as much as possible [2]. Due to the hardware limitation, the resources available at the edge node are far less powerful than cloud servers. Therefore, to achieve a balanced workload between the end devices as well as a reduced communication latency, it is necessary to distribute the calculation tasks between the mobile terminal, the edge node, and the remote cloud server according to the runtime status of the application.

In order to build software in the traditional cloud computing engineering implementation, the engineer should first analyze various functions in the program, identify the module's dependencies, code the software and finally deploy the application in the cloud server to achieve the functioning remote service. Such a classic engineering solution is generally called the server-terminal architecture [1]. However, heterogeneous computation distribution of cloud, edge, and terminal brings new challenges to the traditional computing framework. First, there is a tendency that the software in recent years is becoming more calculation-heavy. With the introduction of complex computing tasks like deep learning algorithms, decomposition may become the main revolution direction of reducing software complexity. Second, it is hard to predict the network environment of the terminal since more mobile devices are introduced into the system. The engineers can no longer assume a stable network condition between the end devices, making it hard to determine what kind of components are most suitable for running on the edge. Third, the types of terminal devices will continue to expand in the future as the Internet of Things (IoT) technologies grow, ranging from smart buttons to smart cars. Their computational performance may vary significantly. Therefore, the new types of software in the edge-computing system are now more computational and communication intensive. Thus, to fully utilize the computational capacity of the devices at each end of the distributed system, software should be able to be decomposed into components to achieve the optimal distribution of overall performance.

Decomposing software into a set of executable components and adjust this partition results dynamically according to the system environment is critical in maximizing the program's overall efficiency. In fact, various types of cloud applications that are widely used nowadays (e.g. the classic MapReduce [17] and deep neural networks [12]) have the improvement potential for software decomposition since it is possible to decompose the data modules and the neural network architectures in these programs. Although there have been many current studies on the algorithms and optimization for partitioning program components under simulated scenarios [15,19], few have done practical implementations and quantitative experiments on the partitioning strategies. Thus, it is essential to implement a system that facilitates the measurements and the performance testing of the current software under the edge-assisted cloud computing scenario.

In this paper, we have proposed and implemented a component-based framework for an edge computing system that allows dynamic partitioning of the software components. Our proposed system can help in further investigations of optimizing the components distribution strategies between the cloud, edge, and terminal devices by measuring the system performance of different types of applications in the deployed settings. Our experiments have also proved the effectiveness of our system by outperforming the traditional cloud-terminal system in the real-world unstable network environments. The outline of the paper is as follows. We reviewed the related works in Sect. 2. The design of our proposed system is shown in Sect. 3. Section 4 presents our implementation of our testing environment. In Sect. 5, we discussed different experiment settings and presented the corresponding results. Section 6 concludes the paper and discusses aspects for future improvements based on this work.

2 Related Work

2.1 Decomposition

Software decomposition is a way to decouple a large system by dividing software into executable components, which is closely related to the program slicing problem [18]. Software decomposition can be classified as functional decomposition [11,13] or micro-services decomposition [7,8]. Functional decomposition decomposes the program into a series of functions (methods) and constructs a working process tree by analyzing the execution process. Micro-services decomposition adopts the theory from Object-Oriented Programming model. In this work, we adopt the programming model from micro-services, deconstructing the applications in our system into different low-coupling instances in software engineering terms.

2.2 Performance Evaluation

Performance evaluation of decomposed software components is a crucial technique for evaluating the execution efficiency of heterogeneous components. Methods can be summarized as model-based analysis [9,11] and statistic prediction [13,20]. The model-based analysis builds a model on the static structure of the program code and estimates the system performance by combing the structure with the real-time execution state of each component. Statistic prediction infers performance evaluation results by maintaining a timeline table of all components at execution. However, the current performance evaluation methods have shown these drawbacks: For the model-based analysis, it is hard to construct a model that can precisely predict the performance. For the statistic prediction, it heavily relies on two assumptions: 1. Components' performance within one application is stable. 2. The consumption of components execution is based on static time complexity analysis. Our previous results [5] have proved that these two assumptions are not correct. Therefore, to address problems in performance evaluation, we have implemented an effective measuring system in our testing framework for component-based software.

2.3 Migration

To support the migration of software between different end devices, current popular solutions include virtual machine migration [9–11] and mobile agent migration [4,5]. Virtual machine migration should build a complete application including the system image. This kind of solution is easy to deploy in the cloud but hard for mobile devices and edge nodes, which is weak in computation ability to support the full capacity of running components. On the contrary, mobile agent solution can migrate corresponding decomposed heterogeneous resources in the components level without consuming much computational resources. In this paper, we adopt Mobile Agent as our migration implementation to studying the component distribution between the cloud, edge, and terminal.

3 System Design

Based on the cloud-terminal architecture proposed in the previous work [5], we continue to design an architecture that incorporates edge computing as follows.

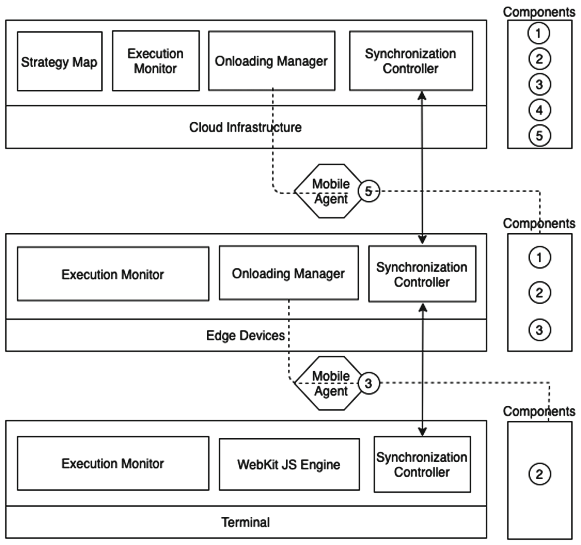


Fig. 1. System Architecture

Figure 1 describes the main components of the proposed system. Overall, the entire architecture can be divided into three layers. The first layer is the Cloud server layer with unlimited computing resources and storage of all the software components. The rectangle on the right represents the component pools of each layer, which is part of the storage that persists the executing code of the corresponding components. The dots with different numbers represent different components. The Edge server connects with the cloud and starts to request components. Then the cloud carries components to the edge's component pool and the terminal's component pool. If the edge detects the existence of a missing component on the terminal side, it can also pass this code to the terminal's component pool. In this process, the component performs inter-component communication through the synchronization controller to prevent unmatched computing status.

3.1 Constructing Components

Execution Monitor is a general system component that exists on each side of the system. It provides monitoring tools for measuring the resource usage of all the components executing on that end, including the runtime CPU cycle or memory usage of components. These data are collected to analyze the effect of the software components distributions. On the terminal-side, the performance recorder(e.g., an FPS recorder) is created to demonstrate the real-time user experience of the application. Since it best indicates the overall system's user satisfaction, we have been adapting this as the criteria for the system QoE in our experiments. *Synchronization Controller* in our system is designed to allow parameter updates in the application. Since in a remotely distributed system, the components are executed separately on different end devices, it is common to have the problem of data getting out of synchronization. Thus, in our system architecture, every component can only communicate with each other through the *Synchronization Controller*. Moreover, the parameter messages in our system need to be compiled and serialized into a JSON string before passing to its destination component as a message. *Onloading Manager* is only implemented in the cloud and the edge system, since all the application codes were originally stored on the cloud and the edge devices. When the system was first started, it would call the *Onloading Manager* to do the first preloading steps to have all the application components stored in the database on the cloud as well as on the edge ends. Since our system does not require any installation of application software on the terminal device, the *Onloading Manager* will load the software components to the terminal before the software was started or running as a backend service as the software is being used in the terminal end. *Onloading manager* achieves components code migration through *Mobile Agent*, a stringified message of the software components that can be passed between ends. The *Strategy Map* in the cloud is a configuration file to guide the distribution of each component. It needs to be loaded in the cloud server's initial stage since the cloud server will be loading the corresponding components to each end in the first preloading process. The dynamic partitioning of the components can be easily achieved by changing the configuration file of the *Strategy Map*. The *WebKit JS Engine* is a native Javascript Engine in browser kernel that is responsible for image rendering computation on the terminal-side.

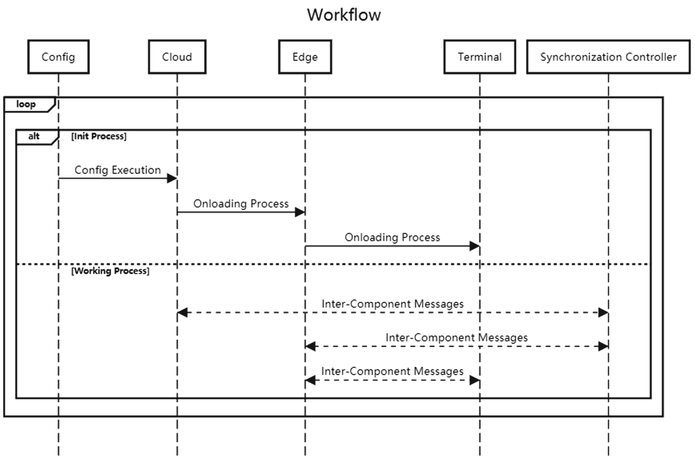


Fig. 2. Workflow

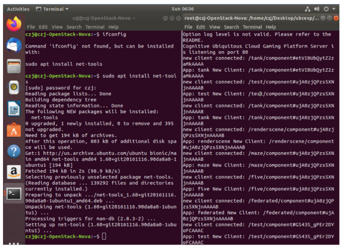
3.2 System Workflow

As shown in Fig. 2, after the application instance starts to run, the entire running process is similar to an infinite state machine. In the initialization phase, the cloud server will first launch the application instance in the cloud server to start its execution. Meanwhile, the cloud server transmits the corresponding component to the Edge server and the terminal server through the *Onloading Manager* according to the pre-specified partition strategy in *Strategy Map*. After the local calculation of each component finishes, each component in the cloud, edge, or terminal will pass through the message passing in the *Synchronization Controller*. In this way, we can ensure the proper execution of the software in this distributed setting.

4 Test Bed Implementation

After evaluating the current system's needs and the experimental goals, we decided to reconstruct the original cloud-terminal system to support offloading to the edge end. JavaScript is currently the only language supported by the browser on all platforms. Using Node.js, a runtime package of JavaScript, we wrote our server end program. This kind of combination allows us to implement the whole project within one programming language. In this setting, the offloaded and migrated computing components can continue its execution in any end without any modification to the code. On the terminal-side, we embedded a Web Kit based browser to parse and execute our JavaScript code so that our system can be well tested on any platform.

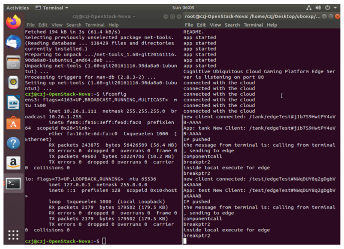
We have been using the JavaScript Socket.IO library to establish our connection between the end devices. The library enables us to build sockets allowing real-time, bidirectional, and event-based communication between the terminal and the servers. In our system implementation, each end device will establish two socket connections when the system is fired up: the cloud-terminal socket, the edge - terminal socket, and the edge-cloud socket. These sockets are responsible for component onloading, connection management, and components message passing. In our system, to enable the migration of the components, we have encapsulated the software components into separated JavaScript files wrapped in a function. The component will first be converted as an executable string in the system, and then use the built-in function *eval* in JavaScript to run corresponding tasks. The demonstration of our test bed is shown in Fig. 3.



```
czj@czj-OpenStack-Nova:~$ ifconfig

Command 'ifconfig' not found, but can be installed
with:

sudo apt install net-tools

czj@czj-OpenStack-Nova:~$ sudo apt install net-tool
s
[sudo] password for czj:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  net-tools
0 upgraded, 1 newly installed, 0 to remove and 395
not upgraded.
Need to get 194 kB of archives.
After this operation, 803 kB of additional disk spa
ce will be used.
Get:1 http://us.archive.ubuntu.com/ubuntu bionic/ma
in amd64 net-tools amd64 1.60+git20161116.90da8a0-1
ubuntu1 [194 kB]
Fetched 194 kB in 2s (98.9 kB/s)
Selecting previously unselected package net-tools.
(Reading database ... 139292 files and directories
currently installed.)
Preparing to unpack .../net-tools_1.60+git20161116.
90da8a0-1ubuntu1_amd64.deb ...
Unpacking net-tools (1.60+git20161116.90da8a0-1ubun
tu1) ...
Processing triggers for man-db (2.8.3-2) ...
Setting up net-tools (1.60+git20161116.90da8a0-1ubu
ntu1) ...
czj@czj-OpenStack-Nova:~$
```

```
Option log level is not valid. Please refer to the
README.
Cognitive Ubiquitous Cloud Gaming Platform Server i
s listening on port 80
new client connected: /tank/component#etVIBUbQytZ2z
aMkAAAA
App: tank New Client: /tank/component#etVIBUbQytZ2z
aMkAAAA
new client connected: /test/component#ujA0zjQPzs5XN
jnAAAAB
App: test New Client: /test/component#ujA0zjQPzs5XN
jnAAAAB
new client connected: /tank/component#ujA0zjQPzs5XN
jnAAAAB
App: tank New Client: /tank/component#ujA0zjQPzs5XN
jnAAAAB
new client connected: /renderscene/component#ujA0zj
QPzs5XNjnAAAAB
App: renderscene New Client: /renderscene/component
#ujA0zjQPzs5XNjnAAAAB
new client connected: /maze/component#ujA0zjQPzs5XN
jnAAAAB
App: maze New Client: /maze/component#ujA0zjQPzs5XN
jnAAAAB
new client connected: /five/component#ujA0zjQPzs5XN
jnAAAAB
App: five New Client: /five/component#ujA0zjQPzs5XN
jnAAAAB
new client connected: /federated/component#ujA0zjQP
zs5XNjnAAAAB
App: federated New Client: /federated/component#ujA
0zjQPzs5XNjnAAAAB
new client connected: /test/component#GS43S_gPEr2DY
oFCAAAC
App: test New Client: /test/component#GS43S_gPEr2DY
oFCAAAC
```

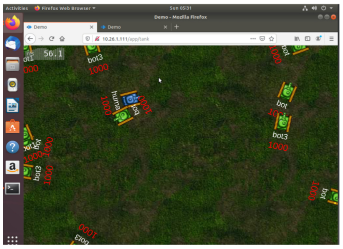
(a) cloud experiment screenshot



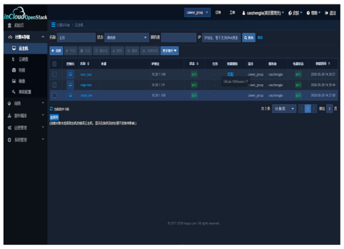
```
Fetched 194 kB in 3s (61.4 kB/s)
Selecting previously unselected package net-tools.
(Reading database ... 138429 files and directories
currently installed.)
Preparing to unpack .../net-tools_1.60+git20161116.
90da8a0-1ubuntu1_amd64.deb ...
Unpacking net-tools (1.60+git20161116.90da8a0-1ubun
tu1) ...
Processing triggers for man-db (2.8.3-2) ...
Setting up net-tools (1.60+git20161116.90da8a0-1ubu
ntu1) ...
czj@czj-OpenStack-Nova:~$ ifconfig
ens3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  m
tu 1500
        inet 10.26.1.111  netmask 255.255.255.0  br
oadcast 10.26.1.255
        inet6 fe80::f816:3eff:fe6d:fac0  prefixlen
64  scopeid 0x20<link>
        ether fa:16:3e:6d:fa:c0  txqueuelen 1000  (
Ethernet)
        RX packets 243875  bytes 56426509 (56.4 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 49603  bytes 10224706 (10.2 MB)
        TX errors 0  dropped 0 overruns 0  carrier
0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host
>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 2179  bytes 179502 (179.5 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2179  bytes 179502 (179.5 KB)
        TX errors 0  dropped 0 overruns 0  carrier
0  collisions 0

czj@czj-OpenStack-Nova:~$
```

```
README.
app started
app started
app started
app started
app started
app started
Cognitive Ubiquitous Cloud Gaming Platform Edge Ser
ver is listening on port 80
connected with the cloud
connected with the cloud
connected with the cloud
connected with the cloud
connected with the cloud
connected with the cloud
new client connected: /tank/edgeTest#j1b7S9HwtPY4uV
B-AAAA
App: tank New Client: /tank/edgeTest#j1b7S9HwtPY4uV
B-AAAA
IP pushed
the message from terminal is: calling from terminal
, sending to edge
componentcall
breakptr2
inside local execute for edge
breakptr2
new client connected: /test/edgeTest#NWqDUY8q2gDgbV
aKAAAB
App: test New Client: /test/edgeTest#NWqDUY8q2gDgbV
aKAAAB
IP pushed
the message from terminal is: calling from terminal
, sending to edge
componentcall
breakptr2
inside local execute for edge
breakptr2
```

(b) edge experiment screenshot



(c) terminal experiment screenshot



(d) OpenStack virtualization platform

Fig. 3. System screenshots

5 Experiment

In this article, our experiment's main purpose is to prove the edge-based system's performance improvement compared with pure cloud-terminal architecture. We adopt the Tank Game Application illustrated in previous work [5] as the main testing benchmark.

The experiment part is constructed as follows. We first conducted single-end measurements, which means all components are all running on either the cloud, the edge, or the terminal device. The first experiment is to measure the effects of an increased computational burden. We have considered two aspects for the computational intensity: the iteration times of each component, and the component quantity in the system. After finishing the necessary measurements of computation ability, we have conducted *intra-cloud*, *intra-edge*, and *intra-terminal* experiments to determine the message passing ability of each ends. They have shown varying degrees of attenuation in FPS as the message length increase of message frequency increases. Combined with the conclusion above, we have tested the edge-based system with cloud-terminal architecture in different network configurations. Finally, we have compared different hardware computing power setups to discover the relationships between the system performance and the computational capacity of the edge device. Based on our measurements, we have proved that in a changing network condition, the edge-based system can perform better than traditional architecture. Moreover, we have found a marginal effect in the reward of increase the resources at edge to increase the system performance.

5.1 Experiment Settings

We have deployed our systems on Ubuntu 18.04 virtual machines created on the OpenStack platform. The cloud system is initially configured with 16-core CPU and 32 GB RAM; The edge system configured with 4-core CPU and 8 GB RAM; The terminal system with 2-core CPU and 4 GB RAM.

With our distributed system, we have mainly tested from two aspects: **Computational Cost** can be adjusted by *Iteration Times*, *component quantity*, and **Communication Cost** can be adjusted by *Communication Frequency* and *Message Length.* In our experiments for **Computational Cost**, no other communication except for those essential for basic inter-component invocation. With this setting, we can analyze the impact of components on computational complexity and communication complexity separately. After measuring the system parameters from the two experiments, we have constructed a hardware-level case study on the impact of **Computing Power Ratio** setting between the cloud, edge, and the terminal devices in this scenario.

Computational Cost: This experiment will consider factors that determine the computational intensity of the components. Considering our system design, we have chosen the *Iteration Number* and *Component Quantity* as our parameters for measurements of the system's performance on calculation tasks. *Iteration Number* is set to be the calculation task number of a single component, which is proportional to computation complexity. *Component quantity* increases the computational pressure by having more components running at the same time.

Iteration Number: To find out the execution characteristics of each end, we have designed our experiment with all the components running in the cloud, edge, and terminal end. We start with a small number of iteration (10 iterations) and then gradually increase the number of iterations to 16000. The result is shown in Fig. 4. From the above measurement cases, we can infer that the higher iteration number will decrease the FPS value in the cloud and edge ends. Cloud outperforms the other ends when the components have a smaller number of iterations. Moreover, cloud and edge execution of high iteration components are less stable comparing to the all-terminal execution, which shows almost no FPS change when the iterations number is getting higher. Generally, when the component has a relatively small size of iteration (e.g., 2000 iterations), it should be offloaded to cloud/edge for execution in order to give the best QoE.

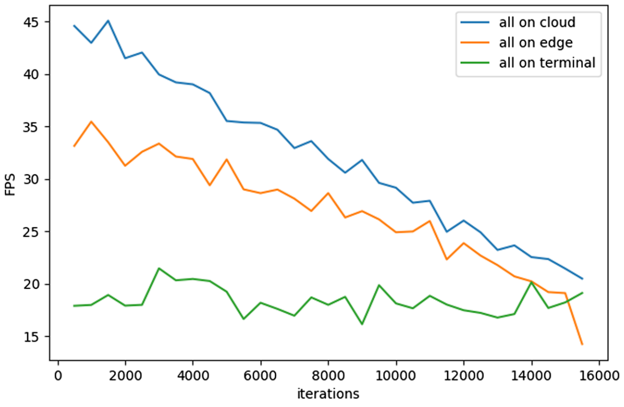


Fig. 4. FPS and iteration numbers at each end

Component Quantity: To further discover the execution characteristics of each end devices, we have experimented with different numbers of components in our system. For all experiments in this series, the iteration number was set to be 2000 iterations, since it was shown in the last experiment that a relatively small iteration is more suitable for cloud/edge execution. Component quantity indicates more components running in a single time period, which can be a measurement of concurrency. Similar to the iteration number experiment, we increase the number of components executing in the single-end system and record their average FPS. The results of Fig. 5 show that an increased number of components leads to a lower FPS value. Moreover, the cloud and edge server are more resistant to the increase in the component quantity than the terminal. Thus, we decided to offload components with more instances to the cloud or the edge side in our further experiments.

Based on the previous two experiments regarding the computing capacity of the end devices in our system, we have constructed a strategy map for further experiments. Components with a large number of instances and a smaller amount of iterations should be distributed to cloud/edge, and a small number of components with large iteration numbers should be left on the terminal.

To verify that our conclusion of different types of components execution on the single end system can show the characteristics of the distributed system, we have further conducted an experiment with components originally executing on the cloud migrating to different end systems and evaluate their performances. The result is shown in Fig. 6. The components with small size of iterations and

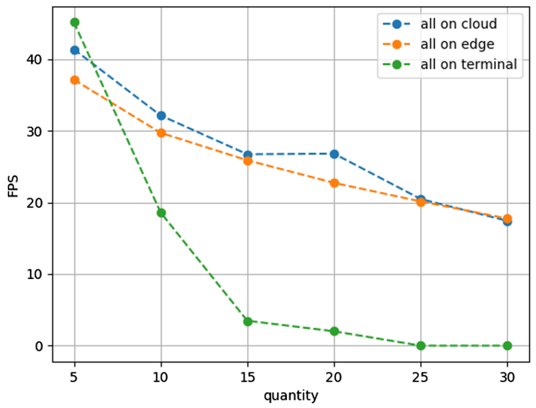


Fig. 5. FPS and quantity numbers at each end

large quantities are originally running at the cloud, yielding a high FPS. The FPS quickly drops when it was migrated to the terminal for execution after 500 time steps. It then further executed for 500 time steps, and then changed to edge for execution. The performance of edge is similar to the cloud in this current setting.

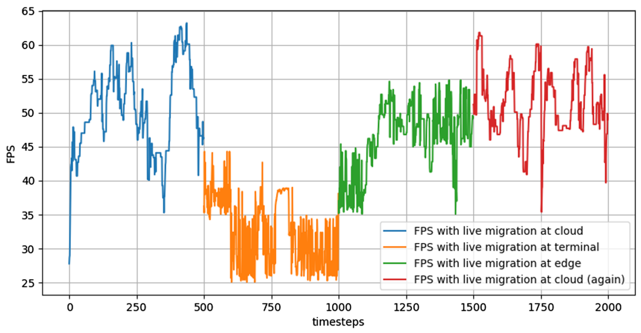


Fig. 6. Live migration of the components

Communication Cost: This experiment will modify the communication frequency and message length under fixed iteration number and component quantity. We aim at discovering the property of both intra-end and inter-end communication in the distributed system.

Intra-end Communications: The three graphs represent *intra-cloud*, *intra-edge* and *intra-terminal* in Fig. 7 respectively. For *intra-cloud* communications, it can be observed that for communication within the cloud, the FPS value will drop as the message length and communication frequency increases. Most of the FPS value is still acceptable for the gaming environment, which means that the cloud end is more suitable for executing components with long and frequent message passing within their instances. For *intra-edge* communications, we have observed a general decrease in overall FPS performance due to the limited computing resources compared to the cloud. However, the execution performance is still at a satisfying level. From the Fig. 7(2), it can be observed that *intra-edge* communication displays a similar property to the *intra-cloud* communication. Thus, the edge is also more suitable for components with larger communication frequencies and message length. For *intra-terminal* communications, it can be observed that transmitting data between components in the terminal greatly reduces the performance of the system. The FPS value dropped the most when the message length is long, and the communication is more frequent compared to the edge/cloud. This implies that the communication within the components at the terminal should be minimized as much as possible.

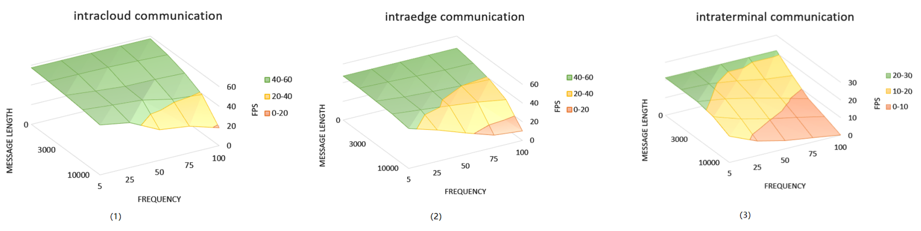


Fig. 7. FPS, message length and communication frequency

In this context, cloud/edge performs better when the message frequency and message length is growing. From the red area portion in the graph, which is the indicator of low-FPS, the *intra-terminal* communication is noticeable worse than the cloud/edge. It also proved that for the browser running on the terminal side, gathering all components is not a good choice, especially when the message communication is increasing. Additionally, we have also observed that comparing to the message length, and the communication frequency will have a more significant influence on the FPS value. For example, in the *intra-edge* setting, if we fix the communication frequency at 50 and only change the message length from 5000 to 10000, the FPS in this case drops from 42.13 to 38.99, whereas if we fix the message length at 5000 and change the communication frequency from 50 to 100, we can observe that the FPS drops to 30.67. This could be caused by the communication overhead of the system in our socket design.

Inter-end Communications: In this experiment, we have conducted experiments on cases where components communicate across heterogeneous environments. Good, fair and bad network were being discussed. Here are related settings: **Good network**, with network delay <30 ms and no package loss in the socket transmission. **Fair network**, with network delay at 60−80 ms and 1% package loss in the socket transmission. **bad network**, with network delay >120 ms and 5% package loss in the socket transmission.

Table 1. Strategy map for network experiments

	cloud_terminal configuration	MEC configuration
Cloud	'bot'*4, 'bot2'*4	'bot' *4
Edge	–	'bot1'*2 ,'bot2'*4
Terminal	'ui','exec','bot1'*2	'ui','exec'

The strategy map we adopted in the experiment is shown in Table 1. There are a total of 12 components running in our testing system, with iteration number 2000, message passing frequency 100, and message length 5000. From the results shown in Fig. 8, we can infer that the network condition has a direct impact on system performance. Under good network conditions, the original cloud-terminal system has the best performance and the highest FPS value due to the better computation capacity in the cloud.

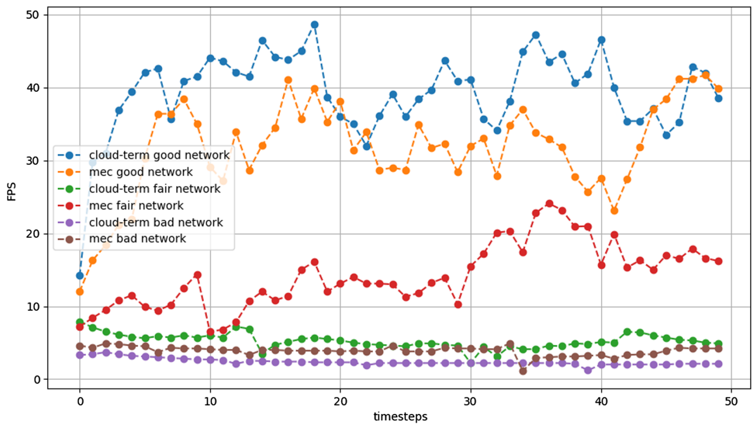


Fig. 8. FPS, message length and communication frequency between ends

When the network between the terminal and the cloud starts to become unstable, the edge-assisted systems perform much stable than the cloud-terminal system. The edge execution of communicative components can fully utilize the high-speed network between the edge and the other ends, resulting in a higher FPS. When the system is under a bad network condition with more package loss between the terminal and the cloud, the performance drops very quickly.

With this experiment, we have shown that with the assistance of edge, the system performance can be greatly improved when there is a fair but unstable network between the mobile device and the cloud. For general case components, our proposed edge-based system generally shows a better performance comparing to the current cloud-terminal system in an environment with network throttling. In component-wise communications, the message length and frequency can impact the overall performance. For our system, we found that the communication frequency will have a more significant influence on the overall FPS than the message length.

Computing Power Ratio: In this experiment, we altered the hardware settings (CPU core number, memory size) to achieve different computing power ratios between different end devices in the edge-assisted cloud computing system. In this experiment, our hypothesis is that increasing the number of resources that we deploy on the edge device will not be giving equal improvement to the system performance. We have adopted the strategy map discussed in the previous experiment settings. We adopted the fair network scenario (80 ms delay, 1% package loss in each transmission) to simulate common network situation in reality without extreme assumption on connection quality. We have tested our system using different sets of hardware. The computing power ratio settings of different experimental groups are shown in Table 2:

Table 2. Hardware configuration table

Cloud CPU cores	Edge CPU cores	Terminal CPU cores
16	8	2
16	4	2
8	4	2
8	2	1
4	2	1

We can infer from Fig. 9 that comparing with the original cloud-terminal configuration, adding edge to the system can have a great improvement on the system FPS. However, if we increase the CPU core number of the edge device from 4 cores to 8 cores, the improvement of FPS is not as significant as expected, which indicates a diminishing marginal utility in the hardware configurations for the edge end. This might be caused by the lack of edge offloading. The most computational intense components should be directed to the cloud for quicker computation. In this case, having a more powerful cloud will give a larger improvement to system performance.

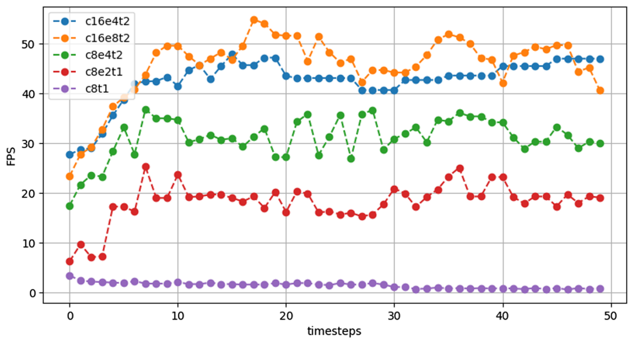


Fig. 9. FPS of different hardware settings w.r.t time steps

6 Conclusion

In this paper, we have proposed and implemented an edge-based dynamic partition testing platform. The proposed system can function as a monitoring tool for the execution of distributed applications in edge computing architecture. Our experiments on the system revealed the characteristics of program execution in different end devices. We have also shown that by properly offloading different types of program components, the edge-based system can improve the overall performance and is more resistant to the fluctuating network. For future works, we will focus on improving the current system from the following perspectives:

1. We shall realize automated component transferring between ends with the real-time data using cognitive algorithms.
2. We shall extend the current single edge system to include multiple edges in our architecture.

References

1. Balan, R.K., Satyanarayanan, M., Park, S.Y., Okoshi, T.: Tactics-based remote execution for mobile computing. In: Proceedings of the 1st International Conference on Mobile Systems, Applications and Services, pp. 273–286 (2003)
2. Bilal, K., Erbad, A.: Edge computing for interactive media and video streaming. In: 2017 Second International Conference on Fog and Mobile Edge Computing (FMEC), pp. 68–73. IEEE (2017)
3. Buyya, R., Yeo, C.S., Venugopal, S.: Market-oriented cloud computing: vision, hype, and reality for delivering it services as computing utilities. In: 2008 10th IEEE International Conference on High Performance Computing and Communications, pp. 5–13. IEEE (2008)
4. Cai, W., Chan, H.C., Wang, X., Leung, V.C.: Cognitive resource optimization for the decomposed cloud gaming platform. IEEE Trans. Circ. Syst. Video Technol. **25**(12), 2038–2051 (2015)
5. Cai, W., Chi, Y., Zhou, C., Zhu, C., Leung, V.C.: Ubcgaming: ubiquitous cloud gaming system. IEEE Syst. J. **12**(3), 2483–2494 (2018)

6. Chen, X., Jiao, L., Li, W., Fu, X.: Efficient multi-user computation offloading for mobile-edge cloud computing. IEEE/ACM Trans. Network. **24**(5), 2795–2808 (2015)
7. Chuang, S.N., Chan, A.T., Cao, J.: Dynamic service composition for wireless web access. In: Proceedings International Conference on Parallel Processing, pp. 429–436. IEEE (2002)
8. Chuang, S.N., Chan, A.T., Cao, J., Cheung, R.: Dynamic service reconfiguration for wireless web access. In: Proceedings of the 12th International Conference on World Wide Web, pp. 58–67 (2003)
9. Chun, B.G., Ihm, S., Maniatis, P., Naik, M., Patti, A.: Clonecloud: elastic execution between mobile device and cloud. In: Proceedings of the Sixth Conference on Computer Systems, pp. 301–314 (2011)
10. Chun, B.G., Maniatis, P.: Dynamically partitioning applications between weak devices and clouds. In: Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond, pp. 1–5 (2010)
11. Cuervo, E., et al.: Maui: making smartphones last longer with code offload. In: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, pp. 49–62 (2010)
12. Fadlullah, Z.M., et al.: State-of-the-art deep learning: evolving machine intelligence toward tomorrow's intelligent network traffic control systems. IEEE Commun. Surv. Tutorials **19**(4), 2432–2455 (2017)
13. Kosta, S., Aucinas, A., Hui, P., Mortier, R., Zhang, X.: Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In: 2012 Proceedings IEEE Infocom, pp. 945–953. IEEE (2012)
14. Kumar, K., Lu, Y.H.: Cloud computing for mobile users: can offloading computation save energy? Computer **43**(4), 51–56 (2010)
15. Liu, J., Zhang, Q.: Adaptive task partitioning at local device or remote edge server for offloading in mec. arXiv preprint arXiv:2002.04858 (2020)
16. Rodrigues, T.G., Suto, K., Nishiyama, H., Kato, N.: A PSO model with VM migration and transmission power control for low service delay in the multiple cloudlets ECC scenario. In: 2017 IEEE International Conference on Communications (ICC), pp. 1–6. IEEE (2017)
17. Slagter, K., Hsu, C.H., Chung, Y.C.: An adaptive and memory efficient sampling mechanism for partitioning in mapreduce. Int. J. Parallel Program. **43**(3), 489–507 (2015)
18. Weiser, M.: Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. Ph.D. thesis, University of Michigan (1979)
19. Yu, S., Wang, X., Langar, R.: Computation offloading for mobile edge computing: a deep learning approach. In: 2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC), pp. 1–6. IEEE (2017)
20. Zhang, X., Kunjithapatham, A., Jeong, S., Gibbs, S.: Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing. Mob. Netw. Appl. **16**(3), 270–284 (2011)