Received September 7, 2018, accepted September 27, 2018, date of publication October 5, 2018, date of current version October 29, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2874057

Performance Analysis of Packets Offloading Scheme Based on Software-Defined Open HetNets Platform

GUANGZHONG LIU AND JIANXIN JIA
College of Information Engineering, Shanghai Maritime University, Shanghai 201306, China
Corresponding author: Jianxin Jia (jmakg23@163.com)

ABSTRACT Following the huge popularization of smartphones and the ensuing explosion of mobile data traffic, cellular networks (LTE) are currently overloaded and this is foreseen to worsen in the near future. While operators continue to rely on their cellular networks to provide wide-area coverage, they are eager to find complementary alternatives to ease the mobile data traffic pressure, especially in areas where subscriber density is very high. Recently, different solutions have been proposed to overcome this problem. Among those solutions, WiFi offloading is the most promising technique for addressing the mobile data explosion issue. However, the LTE-WiFi heterogeneous (HetNets) feature not only complicates the traffic offloading scheme design, but also imposes significant challenges in system-level simulations and evaluations. Lack of innovations in offloading schemes and in simulation techniques will no doubt slow down the evolving and standardization process of next generation cellular systems. Therefore, in this paper, based on the software-defined open HetNets platform, we propose an analytical model to analyze the dual path packets offloading scheme in realistic LTE-WiFi HetNets scenario. Specifically, we first construct the software-defined open HetNets platform. Based on the open HetNets platform, we propose an analytical model for packets offloading in LTE/WiFi HetNets scenario, and derive the mean transmission delay, offloading efficiency, and other metrics of interest, as a function of the key network parameters. The offloading experiment results have verified the accuracy of our analytical model, which shed new light on key aspects of the offloading scheme in LTE-WiFi HetNets.

INDEX TERMS LTE-WiFi HetNets, analytic model, offloading scheme, open HetNets platform, software-defined.

I. INTRODUCTION

We have witnessed an unbelievable growth in telecommunication network traffic, particularly in the mobile network sector. In a recent report, CISCO has claimed that the global mobile data traffic has reached 2.5 exabytes per month at the end of 2014, and it will continue to grow to 24.3 exabyte per month by 2019. This rapid data traffic growth is driven by the proliferation of wireless smart devices and the popularity of media-rich wireless applications [1].

The explosive growth of data traffic brings a great burden to the cellular network (especially in metro areas) and causes a significant degradation of user experience. To alleviate this issue, many new techniques have been introduced for long-term evolution (LTE) and LTE-Advanced networks, such as massive multiple-input multiple-output (MIMO), Heterogeneous Networks (HetNets) with WLANs (i.e., LTE-WiFi HetNets scenario), direct device-to-device communications, etc. Among these cutting-edge techniques [2], WiFi offloading, which is a cost-effective wireless access technology, is considered as a promising solution for addressing the mobile data explosion problem, in which the mobile data traffic is offloaded through WiFi networks in spatially overlapped WiFi and cellular networks [3].

In general, the state of the art WiFi offloading techniques can be classified into three main types, which include the opportunistic offloading, multipath offloading, and the delayed offloading [4]. For the first offloading type, as shown in Fig.1 (1), data offloading is conducted only when a user opportunistically connects to WiFi networks. For the second offloading type, as depicted in Fig.1 (2), two packets transmission paths are maintained, the packets are transmitted seamlessly across two network interfaces at the same time by

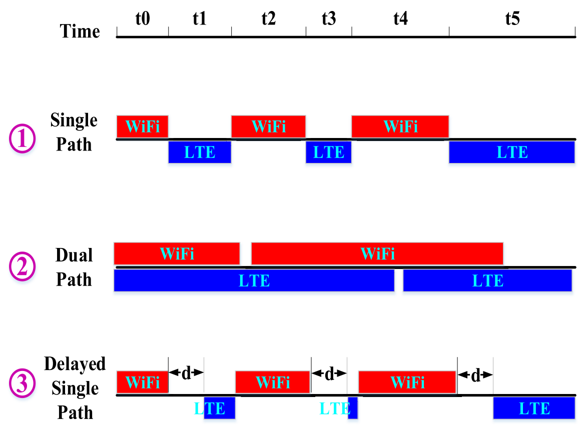


FIGURE 1. Illustrations of three WiFi offloading techniques.

means of simultaneous connections. For the third offloading type, as illustrated in Fig.1 (3), when the WiFi signal coverage is unavailable, packets transmission is delayed by adding a delayed period of d. During the delay period, the data traffic is offloaded to the WiFi network once the connection becomes available. However, if no connection is available during the delayed period, the cellular network connection is constructed to transmit the packets [5]. We can see that the first offloading type is the special case of the second offloading type and the third offloading type is the special case of the first offloading type. In this paper, we mainly focus on the first type offloading technique and the second type offloading technique in particular.

Fig.2 illustrates the typical WiFi offloading scenario in LTE-WiFi HetNets networks where WiFi Access Points (APs) and cellular eNBs are sparsely deployed. Note that some APs deployed within the cellular coverage and some APs are overlaid with the cellular cells. As previously mentioned, the dual-path offloading and the single path offloading techniques are considered in this paper. The detailed procedures of each offloading technique can be described as follows.

Assume a scenario where a UE starts to establish a data session to download/upload a file from/to the Internet at time$t$through the cellular networks (i.e., the blue circle). As UE moving, for the single path packets offloading, if a WLAN is available (i.e., the red circle), it establishes the data session only through the WiFi network. While for the dual-path offloading, when the UE moves into the overlaid areas, it simultaneously uses the LTE-WiFi network immediately [3]. Several WiFi offloading studies have been proposed to describe the above-mentioned offloading techniques in a mathematical way and have been validated by the network simulators. However, those simulators were implemented based on partial 3GPP standard and partial 802.11 standard for the sake of simplicity. In addition, those simulators use the virtual Base station (include eNB and WiFi AP) and virtual UE to conduct the LTE-WiFi HetNets offloading experiment. Therefore, the experiment results obtained from those simulators are not accurate when it comes to the real scenario.

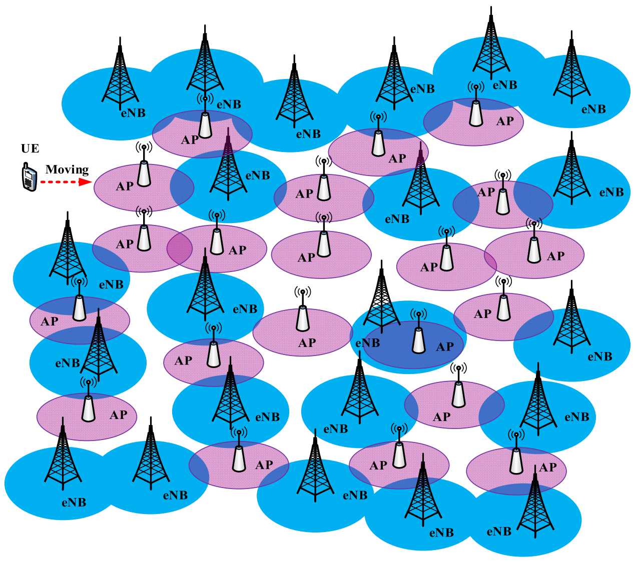


FIGURE 2. The LTE-WiFi HetNets scenario.

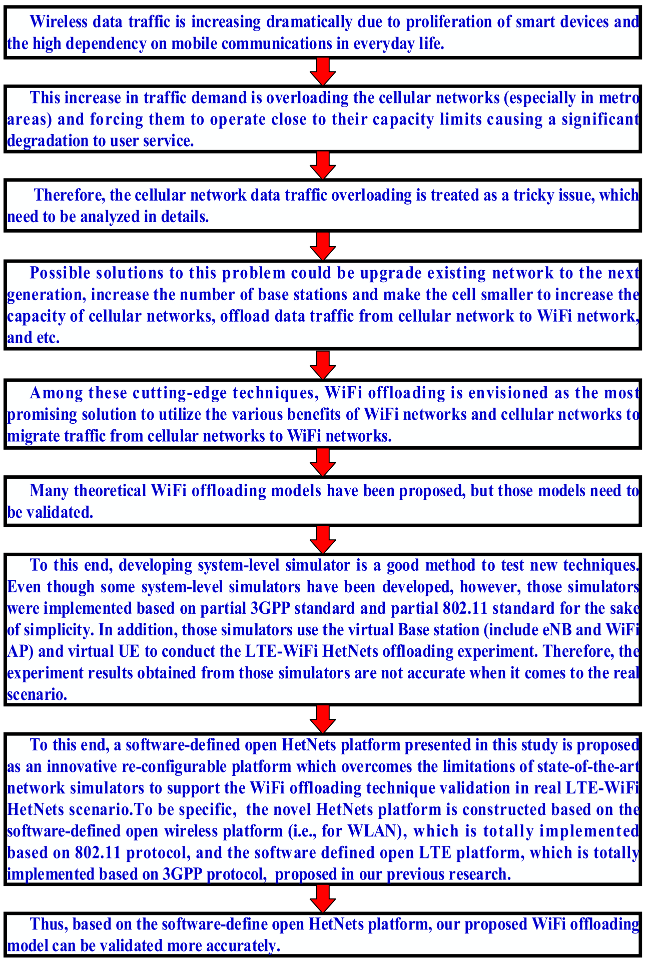
A. MOTIVATION

To this end, a software-defined open HetNets platform presented in this study is proposed as an innovative re-configurable platform which overcomes the limitations of state-of-the-art network simulators to support the WiFi offloading technique validation in real LTE-WiFi HetNets scenario. To be specific, the novel HetNets platform is constructed based on the software-defined open wireless platform (i.e., for WLAN), which is totally implemented based on 802.11 protocol, and the software defined open LTE platform, which is totally implemented based on 3GPP protocol, proposed in our previous research [6], [7]. In order to make our paper's main motivation more clearly, we utilize the motivation flow shown in Fig.3 to summary what we have presented.

B. OUR CONTRIBUTIONS

To sum up, the goal of this paper is first to construct the software-defined open HetNets platform, and then to analyze and characterize the performance of the packets offloading scheme in LTE-WiFi HetNets scenario. In particular, the main contributions of our work can be summarized as follows:

- Based on the software-defined open WiFi platform [6] and the software-defined open LTE platform 7] proposed in our previous research, we propose a software-defined open HetNets platform which is composed by the enhanced EPC (Evolved Packet Core), the software-defined open WiFi platform, and the software-defined open LTE platform. Specifically, we first construct an initial version enhanced EPC including four modules which are the Virtual Upper MAC module, the Offloading Engine module, and two Offloading Observing Window modules. We then do some optimizations for the initial version and evolve it



Wireless data traffic is increasing dramatically due to proliferation of smart devices and the high dependency on mobile communications in everyday life.

This increase in traffic demand is overloading the cellular networks (especially in metro areas) and forcing them to operate close to their capacity limits causing a significant degradation to user service.

Therefore, the cellular network data traffic overloading is treated as a tricky issue, which need to be analyzed in details.

Possible solutions to this problem could be upgrade existing network to the next generation, increase the number of base stations and make the cell smaller to increase the capacity of cellular networks, offload data traffic from cellular network to WiFi network, and etc.

Among these cutting-edge techniques, WiFi offloading is envisioned as the most promising solution to utilize the various benefits of WiFi networks and cellular networks to migrate traffic from cellular networks to WiFi networks.

Many theoretical WiFi offloading models have been proposed, but those models need to be validated.

To this end, developing system-level simulator is a good method to test new techniques. Even though some system-level simulators have been developed, however, those simulators were implemented based on partial 3GPP standard and partial 802.11 standard for the sake of simplicity. In addition, those simulators use the virtual Base station (include eNB and WiFi AP) and virtual UE to conduct the LTE-WiFi HetNets offloading experiment. Therefore, the experiment results obtained from those simulators are not accurate when it comes to the real scenario.

To this end, a software-defined open HetNets platform presented in this study is proposed as an innovative re-configurable platform which overcomes the limitations of state-of-the-art network simulators to support the WiFi offloading technique validation in real LTE-WiFi HetNets scenario.To be specific, the novel HetNets platform is constructed based on the software-defined open wireless platform (i.e., for WLAN), which is totally implemented based on 802.11 protocol, and the software defined open LTE platform, which is totally implemented based on 3GPP protocol, proposed in our previous research.

Thus, based on the software-define open HetNets platform, our proposed WiFi offloading model can be validated more accurately.

FIGURE 3. The motivation flow of why introducing the SD-based HetNets platform for the WiFi offloading issue.

to the optimal version. Finally, we conduct the pressure testing experiment for the HetNets platform to check out the maximum throughput that can be supported by it.

- Based on the software-defined open HetNets platform, we construct a simple scenario where the UE can choose between the WiFi and the LTE technology, and derive general formulas, as well as simpler approximations, for the expected transmission delay and offloading efficiency.
- Based on the software-defined open HetNets platform, the accuracy of the analytical results is verified through the packets offloading experiment under real LTE/WiFi HetNets scenario. Specifically, we validate our model in scenarios where most parameters of interest are taken from real measured data. The analytical model proposed in this paper can provide the operator with a powerful tool for tuning the system parameter in such a way as to satisfy the users' transmission requirements while simultaneously maximizing the network capacity in the LTE/WiFi HetNets scenario.

C. OUTLINE OF THE PAPER

The remainder of this paper is organized as follows. We briefly review the related works and the state of the art in Section II. In section III, the construction process of the software defined open HetNets platform is described. In section IV, based on the software-defined open HetNets platform, the packets offloading scheme with closed-form expressions is derived. Section V provides experiment evaluations with verifications to the analytical results. Finally, Section VI provides concluding remarks and future directions.

II. RELATED WORK

To alleviate the traffic burden of cellular networks, mobile operators started adopting data offloading strategies. These solutions involve diverting part of the data traffic from the cellular network, to an alternative path. In many cases, WiFi is used to bear this traffic, because there are already a large number of WiFi access points deployed at homes and throughout cities (e.g., the scenario shown in Fig.2). These access points could be owned by the mobile operator itself or could be accessed upon agreements between operators and other businesses. Thus, researchers and companies have devised several mobile data offloading schemes using WiFi. These solutions usually use only one of simultaneously available networks, which can be categorized as single-path schemes. As an alternative approach, data offloading can also be performed using multipath schemes. In this case, several networks can be used simultaneously. However, this functionality cannot be implemented using current standard Internet protocols. Consequently, either new protocols or extensions to existing ones must be considered [8].

According to the survey reported in Rebecchi *et al.* [9], the issue of offloading IP flows originally via cellular networks to intermittently-connected WiFi networks has gained great attention in recent years. There have been considerable research efforts to exploit the offloading schemes to migrate data traffic from cellular networks toWiFi networks [10].

A quantitative study on offloading cellular traffic to WiFi networks has been presented in [11]. Bennis *et al.* [12] have developed a distributed cross-system learning framework to improve the cellular throughput by offloading data traffic to WiFi networks. Lee *et al.* [11] conducted experiments that capture 100 smartphone users' WiFi usage patterns to investigate the amount of traffic that can be offloaded. Gao *et al.* [13] analyzed the interaction for the amount of offloading data and the respective compensations among one cellular network operator and AP owners by using the Nash bargaining theory. Han *et al.* [14] propose to exploit opportunistic WiFi communications for information spreading in social networks. Their study is based on determining the minimum number of users that are able to reduce maximally the amount transmitted through the cellular network. The LTE offloading into WiFi direct is subject of study in [15]. The work in [16] is mainly concerned with studying the conditions under which rate coverage is maximized, for random deployment of APs belonging to different networks. Contrary to most of the other works, Lee *et al.* [17] consider the situation in which cellular operators pay for using the AP from third parties. They use game theory to consider different issues, such as the amount of data and money a cellular operator should pay for utilizing the APs. In [18], a solution for mobile data offloading between 3GPP and non-3GPP (e.g., WiFi) access networks is presented. A Wi-Fi based mobile data offloading architecture that target the energy efficiency for smartphones was presented in [19]. An interesting work on determining the number of Wi-Fi APs that need to be deployed in order to achieve a QoS is presented in [20].

Although all the above-mentioned schemes demonstrated to work, none of them can use multiple interfaces simultaneously. Multipath transmission schemes can utilize multiple interfaces at the same time. This is especially important when ongoing transmissions confront changes in their currently used interface. With current Internet protocols, the connection will be interrupted and transmitted data will be likely lost. To avoid this interruption, and leverage the use of multiple interfaces, several protocols were proposed [8]. Iyengar *et al.* [21] proposed Concurrent Multipath Transfer (CMT), an extension to the Stream Control Transmission Protocol (SCTP) for simultaneous transmission over multiple interfaces. Koh *et al.* [22] further improved SCTP to support traffic handover among interfaces. Hsieh and Sivakumar [23] developed a protocol called pTCP for multihomed devices. You *et al.* [24] presented an interesting multipath reference architecture. They explained the basics of a generic architecture and multipath protocols design. Paasch *et al.* [25] and Raiciu *et al.* [26] tested a new protocol called Multipath TCP (MPTCP) in a LTE/WiFi scenario. They showed the simultaneous transmission capability of the protocol over real networks and unmodified applications.

To sum up, although the state of the art solutions improves the efficiency of offloading in LTE-WiFi HetNets with respect to the special setting of the offloading related parameters, in these works, however, all users utilize the simulator to validate their LTE-WiFi HetNets offloading analytical model and evaluate the experiment results, which may not suitable when it comes to the realistic scenario. To our best knowledge, the experiment validation for the offloading analytical model, which based on the realistic LTE-WiFi HetNet scenario (i.e., software defined open HetNets platform) rather than based on the simulator, has not been proposed in the literature, which is one of the main contributions of this paper.

III. SOFTWARE-DEFINED OPEN HETNETS PLATFORM

A. SOFTWARE-DEFINED OPEN HETNETS PLATFORM DESIGN

The main purpose of this section is to describe the architecture of the novel open HetNets platform. We first give an overview of those modules integrated in the platform including Virtual Upper MAC module, Offloading Engine module, and Offloading Observing Window module. We then analyze the functionalities and the interactions among the Virtual Upper MAC module, Offloading Engine module and Offloading Observing Window module in depth.

The software architecture of the open HetNets platform is depicted in Fig.4. The picture on the left-hand is the Evolved Packet Core (EPC) and the picture on the right-hand is the software defined user equipment (UE). In the EPC side, the system is divided into two spaces including the kernel space and the user space. As shown in Fig.4, there are two modules in kernel space, one is the Upper Protocol Stack module (i.e., the OS kernel) and the other is the Virtual Upper MAC module. The Upper Protocol Stack module is used for IP packets generation, and the Virtual Upper MAC module is used for receiving the IP packets and then transmitting IP packets to the Offloading Engine module located in user space through Netlink. Besides, there are three modules in user space, which are the user application module, the Offloading Engine module, and the Offloading Observing Window module. The Offloading Engine module is used for receiving the IP packets transmitted from the Virtual Upper MAC module and then deciding which way to transmit those packets to the Offloading Observing Window module. For instance, if the Offloading Engine selects path number one, then it writes the packets to the first transmission ring (i.e., TX1) and sends signal SIGRTMIN to the first Offloading Observing Window module (i.e., Offloading Observing Window1). After Offloading Observing Window1 receives the signal notified by the Offloading Engine, it will read the TX1 and then send the packet to the air through the open LTE platform (i.e., open LTE MAC and open LTE PHY). If Offloading Engine selects path number two, then it writes the packets to the second transmission ring (TX2) and sends signal SIGRTMIN to the second Offloading Observing Window module (i.e., Offloading Observing Window2). After the Offloading Observing Window2 receives the signal notified by the Offloading Engine, it will read the TX2 and then send the packet to the air through the open WiFi platform (i.e., open WiFi MAC and open WiFi PHY). Inversely, if the Offloading Observing Window1 receives packets from port2, then it writes the received packet to the first reception ring (i.e., RX1) and sends signal SIGRTMIN to Offloading Engine to notify it is the right time to read RX1. In the same way, if the Offloading Observing Window2 receives packets from port1, then it writes the received packet to the second reception ring (i.e., RX2) and sends signal SIGRTMIN+1 to Offloading Engine to notify it is the right time to read RX2. Here, we give some brief descriptions to the signal mechanism and the share-memory mechanism (i.e., the light blue rectangle shown in Fig.4). Since the Offloading Engine module and those two Offloading Observing Window modules are all threads when they are in running status, therefore it is suitable to use the signal mechanism to conduct the thread communication. Note that for Offloading Engine module, it is easy to inform those two Offloading Observing Window modules by sending signal SIGRTMIN. However, for the

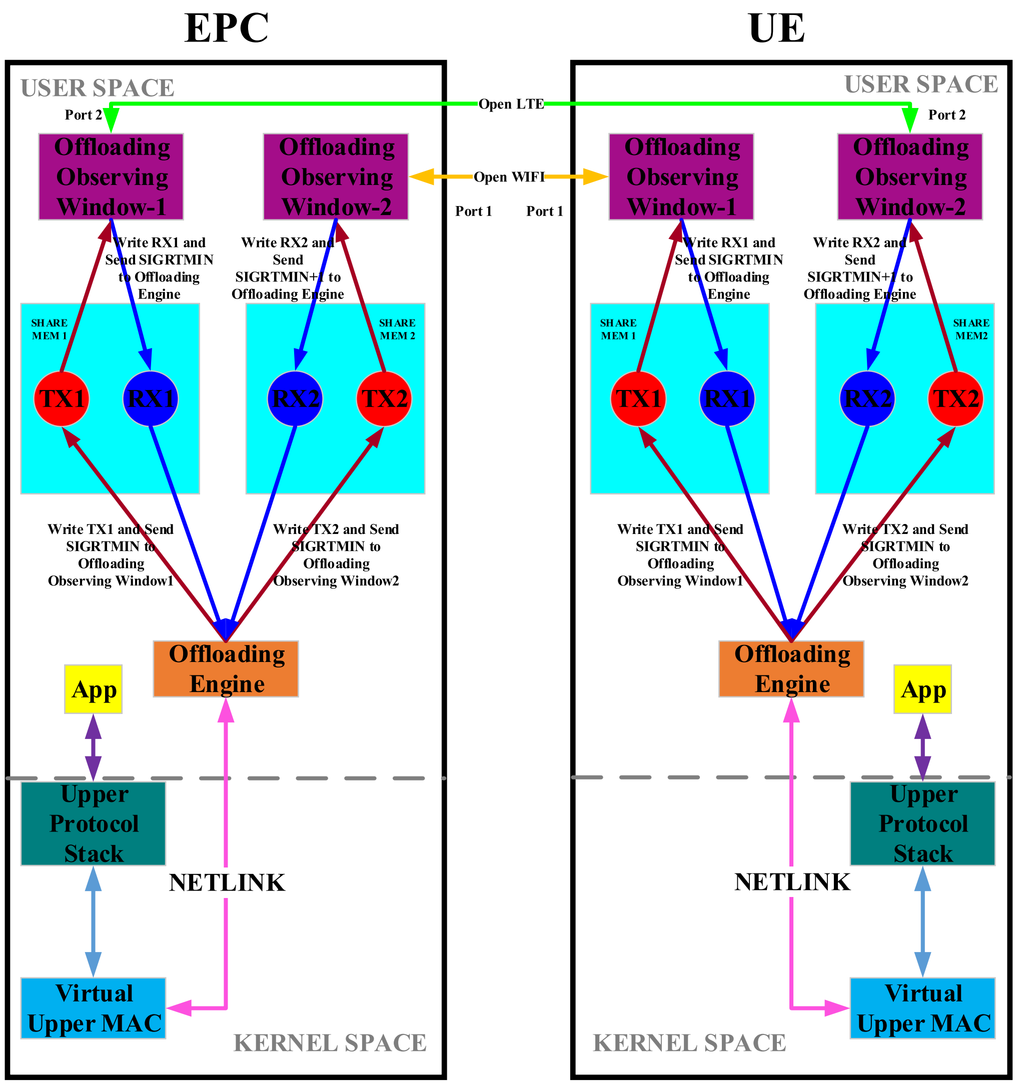

EPC
UE
USER SPACE
Port 2
Open LTE
Open WIFI
Port 1
Offloading Observing Window-1
Offloading Observing Window-2
Write RX1 and Send SIGRTMIN to Offloading Engine
Write RX2 and Send SIGRTMIN+1 to Offloading Engine
SHARE MEM 1
SHARE MEM 2
TX1
RX1
RX2
TX2
Write TX1 and Send SIGRTMIN to Offloading Observing Window1
Write TX2 and Send SIGRTMIN to Offloading Observing Window2
Offloading Engine
App
Upper Protocol Stack
NETLINK
Virtual Upper MAC
KERNEL SPACE

FIGURE 4. The software architecture of the open HetNets platform.

Offloading Engine module, it must use the distinguishable signal to inform the Offloading Engine (i.e., SIGRTMIN and SIGRTMIN+1). The SIGRTMIN and SIGRTMIN+1 are two available signals in Linux systems, and those two signals belong to the available signal set defined in Linux system. The share memory is allocated between the Offloading Engine module and the Offloading Observing Window module, it is used for buffering the sending packets from Offloading Engine and the receiving packets from Offloading Observing Window. In section III.B and section III.C, we will give the detailed descriptions for the signal mechanism and the share-memory mechanism.

The remainder of this section is organized as follows. The Virtual Upper MAC module is illustrated in section III.A. In section III.B, we describe the Offloading Engine module. In section III.C, we present the Offloading Observing Window module. The detailed optimization process for the open HetNets platform is illustrated in section III.D. Finally, section III.E provides some experiment evaluations to the optimized open HetNets platform.

B. VIRTUAL UPPER MAC MODULE

In this section, we first focus on analyzing the formation process of the Upper Protocol Stack packets (i.e., IP packets) and then demonstrating the detailed implementation process of two key functions, which are the packets transmission (Tx) function and the packets reception (Rx) function, located in the Virtual Upper MAC module.

1) PREFACE

As shown in Fig.5, the functional hierarchical structure of the open HetNets platform can be divided into eight layers. This part mainly focuses on the top six layers and the rest two layers (MAC and PHY) have been researched in our previous works (i.e., the software defined open WiFi platform [6] and the software defined open LTE platform [7]).

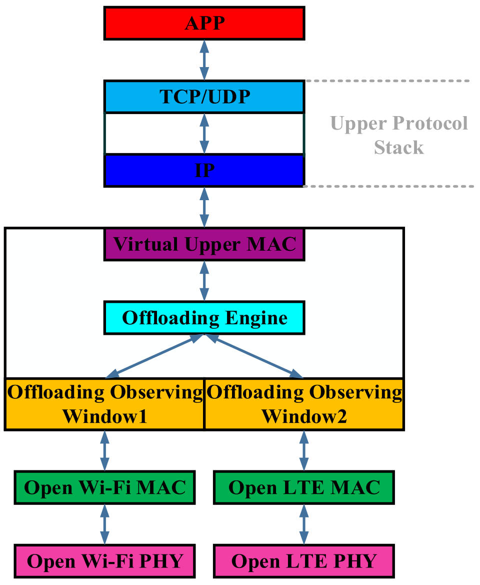


FIGURE 5. **Functional hierarchical structure of the open HetNets platform.**

The Upper Protocol Stack (i.e., the TCP/UDP layer and the IP layer), source of packets transmission and destination of packets reception, is located in the Linux OS (Operating System) kernel. The fourth layer is the Virtual Upper MAC, and the detailed information of Virtual Upper MAC is highlighted in Fig.6. The network device driver in Virtual Upper MAC is a collection of functions that can be compiled as part of the Linux kernel; it is also a "C" program that controls a network device. The network device driver is an important part of Linux network application which follows common network interface. For each network interface, the network device driver uses a device data structure called "struct net_device"

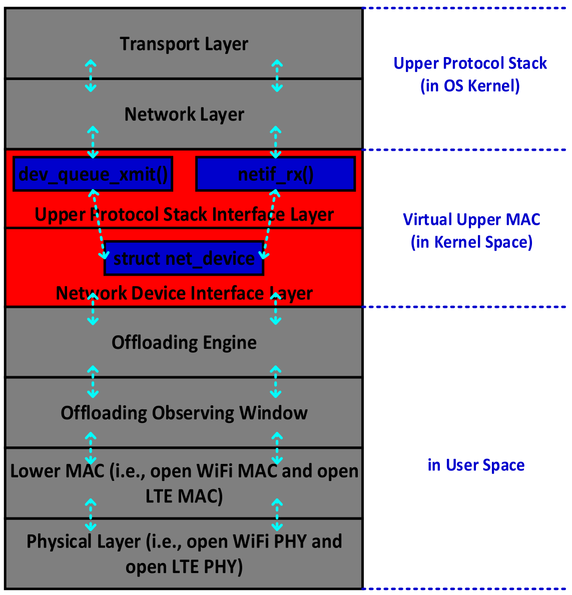

Transport Layer
Network Layer
dev_queue_xmit()
netif_rx()
Upper Protocol Stack Interface Layer
struct net_device
Network Device Interface Layer
Offloading Engine
Offloading Observing Window
Lower MAC (i.e., open WiFi MAC and open LTE MAC)
Physical Layer (i.e., open WiFi PHY and open LTE PHY)
Upper Protocol Stack (in OS Kernel)
Virtual Upper MAC (in Kernel Space)
in User Space

FIGURE 6. **The Virtual Upper MAC module (i.e., the highlighted part).**

to identify it. For generally speaking, the network device is a physical device, such as the Ethernet card. However, in our open HetNets platform, we construct a virtual network device called "openhetnet0", which means there is no need to insert any physical equipment. The "dev_queue_xmit ()" and "netif_rx ()" in Fig.6 are two key functions in the device driver and they are also the interfaces between the Upper Protocol Stack and the Virtual Upper MAC. The dev_queue_xmit () is responsible for packets transmission and the netif_rx () is responsible for packets reception. Therefore, the main work of device driver function layer is to implement these two key functions (i.e., Tx and Rx).

2) THE FORMATION OF THE UPPER PROTOCOL STACK PACKETS

The packets in the Upper Protocol Stack exist in the form of "sk_buff". The "sk_buff" is an OS kernel data type and it is also the central nervous system of packets transmission in Upper Protocol Stack. The formation process of the Upper Protocol Stack packet (i.e., IP packet) is shown from Fig.7 to Fig.9.

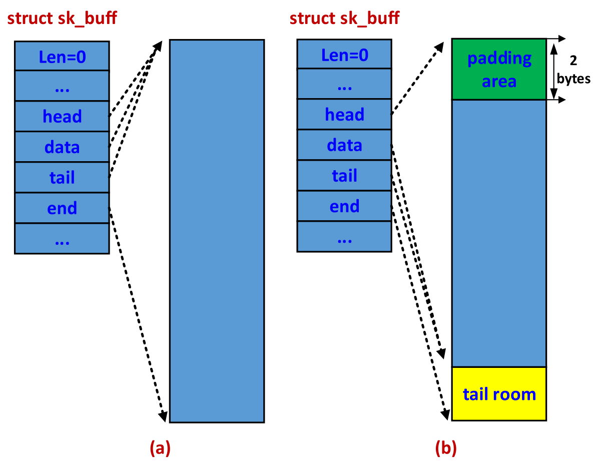


FIGURE 7. **The formation process of the Upper Protocol Stack packet: (a)after "alloc_skb ()" and before "skb_reserve ()"; (b) after "skb_reserve ()".**

Fig.7 (a) demonstrates that the OS kernel first utilize the function "alloc_skb ()" to provide enough data buffer. After the function "alloc_skb ()" is finished, there will be four pointers came out, which are the head pointer, the data pointer, the tail pointer and the end pointer, respectively. Shown in Fig.7 (a), the first three pointers point to the starting position of the data buffer and the last one points to the ending position of the data buffer. Then, the OS kernel uses the function "alloc_reserve ()" to reserve a padding area and a tail room. Shown in Fig.7 (b), the padding area is 2 bytes in length, which is used for alignment. When the function "alloc_reserve ()" fulfil its own mission, the location of data pointer and tail pointer will be changed. In Fig.7, we can also see the Len filed of sk_buff is 0 with the reason that now the data pointer and the tail pointer point to the same position.

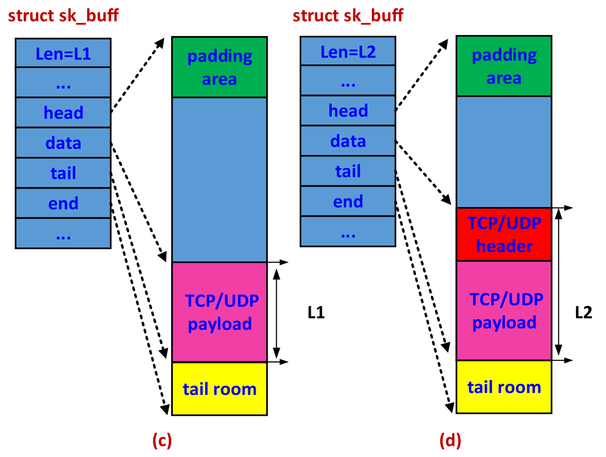


FIGURE 8. **The formation process of the Upper Protocol Stack packet: (c) when copying the TCP/UDP payload to the data buffer; (d) when copying the TCP/UDP header to the data buffer.**

In another words, the buffer length between the data pointer and tail pointer equals to the value of Len filed.

Let $L_{payload}$ refers to the length of the TCP/UDP payload and L_{tuhdr} refers to the length of TCP/UDP header, respectively. As shown in Fig.8, the OS kernel first copies TCP/UDP payload to the pink part and then copies TCP/UDP header to the red part. Once the copy process is done, the location of the data pointer and the value of the Len field will be changed. The corresponding calculations are shown below.

$$\begin{aligned} L_1 &= L_{payload} \\ L_2 &= L_{payload} + L_{tuhdr} \\ sk\_buff \rightarrow data &= data + L_{payload} + L_{tuhdr} \end{aligned} \tag{1}$$

Let L_{iphdr} refers to the length of IP header. As shown in Fig.9, the OS kernel first copies the IP header to the light blue part and then makes the data pointer points to the starting position of the IP header. Once the copy process is done, the location of the data pointer and the value of the Len field will be changed. The corresponding calculations are shown

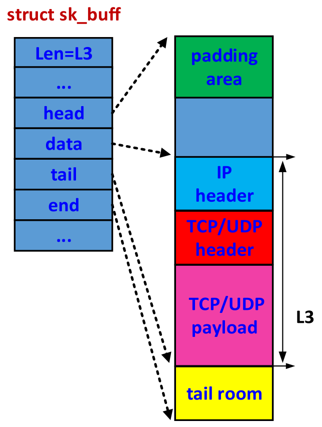


FIGURE 9. **The formation process of the Upper Protocol Stack packet when copying the IP header to the data buffer.**

as below.

$$\begin{aligned} L_3 &= L_{payload} + L_{tuhdr} + L_{iphdr} \\ sk\_buff \rightarrow data &= data + L_{payload} + L_{tuhdr} + L_{iphdr} \end{aligned} \tag{2}$$

Based on the above-mentioned procedures, we can note that the upper protocol packet (IP packet) is made up of three parts which are the light blue part (i.e., IP header), the red part (i.e., TCP/UDP header), and the pink part (i.e., TCP/UDP payload). Till now, we have illustrated the detailed formation principle of the upper protocol stack packets in the OS kernel. In next section, we will give some descriptions for the two key functions located in the Virtual Upper MAC module.

3) TWO KEY FUNCTIONS IN VIRTUAL UPPER MAC

Packets transmission and reception are two key functions in the Linux network device driver, and whether these two processes are good or bad directly affect the Upper MAC overall running quality. As shown in Fig.6, the packets transmission function "dev_queue_xmit ()" is the interface of the packets transmission from the upper layers to the lower layers. The packets reception function "netif_rx ()" is the interface of packets reception from the lower layers to the upper layers.

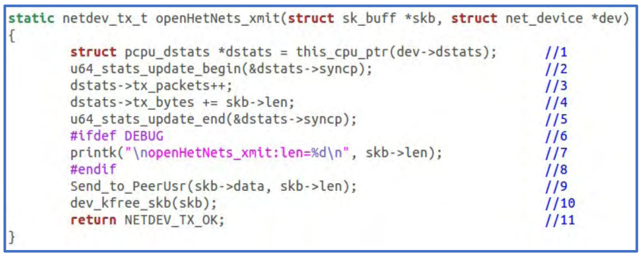
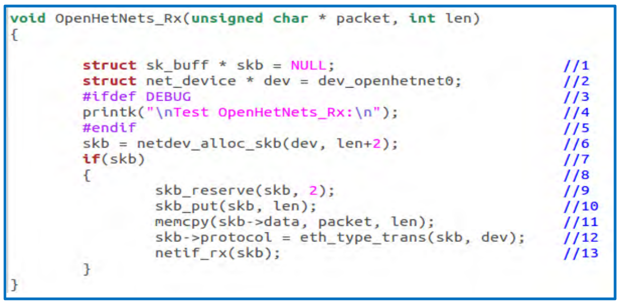

```
static netdev_tx_t openHetNets_xmit(struct sk_buff *skb, struct net_device *dev)
{
        struct pcpu_dstats *dstats = this_cpu_ptr(dev->dstats);      //1
        u64_stats_update_begin(&dstats->syncp);                      //2
        dstats->tx_packets++;                                        //3
        dstats->tx_bytes += skb->len;                                //4
        u64_stats_update_end(&dstats->syncp);                        //5
        #ifdef DEBUG                                                 //6
        printk("\nopenHetNets_xmit:len=%d\n", skb->len);             //7
        #endif                                                       //8
        Send_to_PeerUsr(skb->data, skb->len);                        //9
        dev_kfree_skb(skb);                                          //10
        return NETDEV_TX_OK;                                         //11
}
```

FIGURE 10. **The key code of packets transmission function"openHetNets_xmit ()"**

In our open HetNets platform, we have implemented those two key interface functions (i.e., "openHetNets_xmit ()" and "OpenHetNets_Rx ()") defined in the device driver function layer. Fig.10 depicts the key code of the packets transmission function "openHetNets_xmit ()". Note that in Fig.10, the formal parameters in function header are two pointer variables, of which one points to the arrival upper protocol stack packet and the other points to the virtual network device (i.e., openhetnet0). In Fig.10, from line 1 to line 5, we first do some net device statistic information for CPU. Then, in code line 9, we utilize Netlink, a favorable method of transmitting bio-directional data between kernel space and user space, to transmit the upper protocol stack packets to the Offloading Engine module.

The function "OpenHetNets()" in Fig.11 is used for packets reception, the first parameter in the function header is an unsigned char pointer points to the packet transmitted from Offloading Engine module, and the second parameter indicates the length of received packet. In code line 1, we define a "struct sk_buff" type pointer variable "skb" and let this pointer points to NULL at first. In code line 2, we define a



```
void OpenHetNets_Rx(unsigned char * packet, int len)
{
          struct sk_buff * skb = NULL;                                 //1
          struct net_device * dev = dev_openhetnet0;                   //2
          #ifdef DEBUG                                                 //3
          printk("\nTest OpenHetNets_Rx:\n");                          //4
          #endif                                                       //5
          skb = netdev_alloc_skb(dev, len+2);                          //6
          if(skb)                                                      //7
          {                                                            //8
                    skb_reserve(skb, 2);                               //9
                    skb_put(skb, len);                                 //10
                    memcpy(skb->data, packet, len);                    //11
                    skb->protocol = eth_type_trans(skb, dev);          //12
                    netif_rx(skb);                                     //13
          }
}
```

FIGURE 11. The Key code of packets reception function "OpenHetNets_Rx ()".

"struct net_device" type pointer variable "dev" and let this pointer points to the virtual net device "dev_openhetnet0". In code line 6, we utilize the function "netdev_alloc_skb ()" to allocate a memory buffer and let the pointer "skb" points to it. If the allocation process is success, based on the principle shown in Fig.7, we use function "skb_reserve ()" to reserve a 2 bytes padding area and a tail room. Based on the above-mentioned procedures, from code line 10 to code line 12, we make some assignments for three vital member fields defined in "struct sk_buff" (i.e., skb->protocol, skb->data, skb->len), which must be correctly filled up before the "skb" submits to the upper layers. Finally, in code line 13, we use the function "netif_rx (skb) (i.e., the interface function shown in Fig.6)" to submit the received packet to the upper protocol stack.

C. OFFLOADING ENGINE MODULE

The function caller diagram in Offloading Engine module is shown in Fig.12. Note that the main function calls other six functions when in running status. The code files "signalhandler.h" and "signalhandler.c" are used to conduct the signal notification mechanism between the Offloading Engine module and Offloading Observing Window module, the code files "sharemem.h" and "sharemem.c" are used to process the share-memory mechanism between the Offloading Engine module and Offloading Observing Window module, and the code files "netlink.h" and "netlink.c" are responsible for the communication between the Virtual Upper MAC module located in kernel space and the Offloading Engine module located in user space. In order to illustrate this module's logical function more clearly, in the next, we give the detailed descriptions for each code file. Besides, for the convenience of easy understanding, we can interpret the Offloading Engine module as the virtual lower MAC module and the Offloading Observing Window module as the virtual PHY module in the rest of this section.

The main function of virtual lower MAC module (i.e., Offloading Engine module) is shown in Fig.13.

As illustrated in Fig.13, code line 1, we first use function "getpid ()" to obtain the virtual lower MAC thread ID (i.e., LowMACPid). Then in code line 3, we utilize function "Register_SignalHandler ()" to register two signals

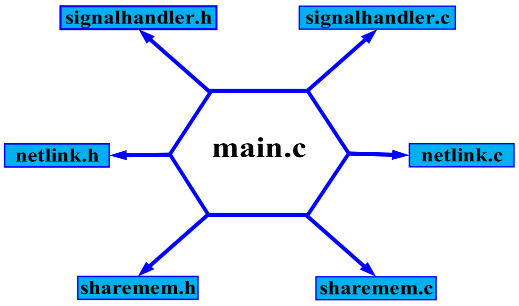
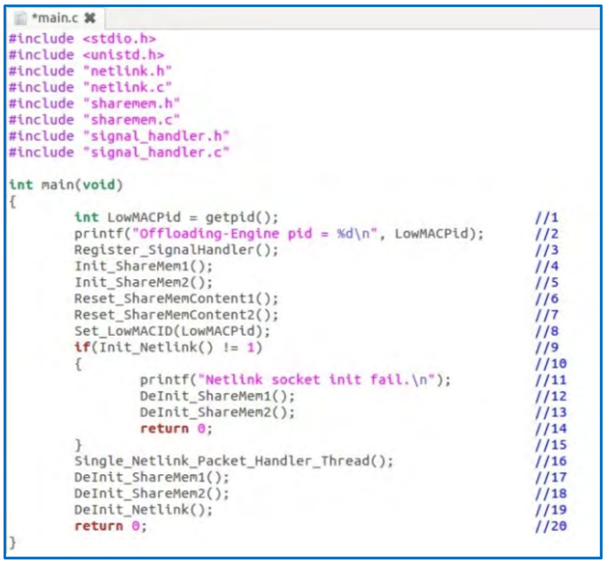


FIGURE 12. The function caller diagram in Offloading Engine module.

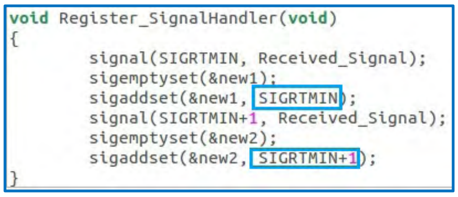


```
*main.c
#include <stdio.h>
#include <unistd.h>
#include "netlink.h"
#include "netlink.c"
#include "sharemem.h"
#include "sharemem.c"
#include "signal_handler.h"
#include "signal_handler.c"

int main(void)
{
        int LowMACPid = getpid();                                      //1
        printf("Offloading-Engine pid = %d\n", LowMACPid);             //2
        Register_SignalHandler();                                      //3
        Init_ShareMem1();                                              //4
        Init_ShareMem2();                                              //5
        Reset_ShareMemContent1();                                      //6
        Reset_ShareMemContent2();                                      //7
        Set_LowMACID(LowMACPid);                                       //8
        if(Init_Netlink() != 1)                                        //9
        {                                                              //10
                printf("Netlink socket init fail.\n");                 //11
                DeInit_ShareMem1();                                    //12
                DeInit_ShareMem2();                                    //13
                return 0;                                              //14
        }                                                              //15
        Single_Netlink_Packet_Handler_Thread();                        //16
        DeInit_ShareMem1();                                            //17
        DeInit_ShareMem2();                                            //18
        DeInit_Netlink();                                              //19
        return 0;                                                      //20
}
```

FIGURE 13. The main function in Offloading Engine module.

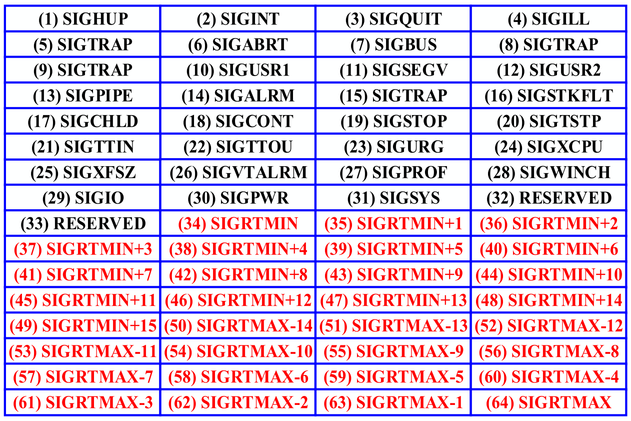
(i.e., the received signals transmit from virtual PHY thread) used by virtual lower MAC thread. The definition of "Register_SignalHandler ()" is included in code file "signalhandler.c" and it is shown in Fig.14.



```
void Register_SignalHandler(void)
{
        signal(SIGRTMIN, Received_Signal);
        sigemptyset(&new1);
        sigaddset(&new1, SIGRTMIN);
        signal(SIGRTMIN+1, Received_Signal);
        sigemptyset(&new2);
        sigaddset(&new2, SIGRTMIN+1);
}
```

FIGURE 14. The definition of function "Register_SignalHandler ()".

In Fig.14, the function "signal ()" first register the signal SIGRTMIN as the received signal of virtual lower MAC thread. Then, the function "sigemptyset ()" empty the original signal set in virtual lower MAC thread and initialize the new registered signal. Finally, function "sigaddset ()" add the new registered signal to the signal set of virtual lower MAC thread. Here, we give some explanations to SIGRTMIN. As shown in Fig.15, we give all the signals defined in Linux system. Note that there are 64 signals in Fig.15, however,



(1) SIGHUP	(2) SIGINT	(3) SIGQUIT	(4) SIGILL
(5) SIGTRAP	(6) SIGABRT	(7) SIGBUS	(8) SIGTRAP
(9) SIGTRAP	(10) SIGUSR1	(11) SIGSEGV	(12) SIGUSR2
(13) SIGPIPE	(14) SIGALRM	(15) SIGTRAP	(16) SIGSTKFLT
(17) SIGCHLD	(18) SIGCONT	(19) SIGSTOP	(20) SIGTSTP
(21) SIGTTIN	(22) SIGTTOU	(23) SIGURG	(24) SIGXCPU
(25) SIGXFSZ	(26) SIGVTALRM	(27) SIGPROF	(28) SIGWINCH
(29) SIGIO	(30) SIGPWR	(31) SIGSYS	(32) RESERVED
(33) RESERVED	(34) SIGRTMIN	(35) SIGRTMIN+1	(36) SIGRTMIN+2
(37) SIGRTMIN+3	(38) SIGRTMIN+4	(39) SIGRTMIN+5	(40) SIGRTMIN+6
(41) SIGRTMIN+7	(42) SIGRTMIN+8	(43) SIGRTMIN+9	(44) SIGRTMIN+10
(45) SIGRTMIN+11	(46) SIGRTMIN+12	(47) SIGRTMIN+13	(48) SIGRTMIN+14
(49) SIGRTMIN+15	(50) SIGRTMAX-14	(51) SIGRTMAX-13	(52) SIGRTMAX-12
(53) SIGRTMAX-11	(54) SIGRTMAX-10	(55) SIGRTMAX-9	(56) SIGRTMAX-8
(57) SIGRTMAX-7	(58) SIGRTMAX-6	(59) SIGRTMAX-5	(60) SIGRTMAX-4
(61) SIGRTMAX-3	(62) SIGRTMAX-2	(63) SIGRTMAX-1	(64) SIGRTMAX

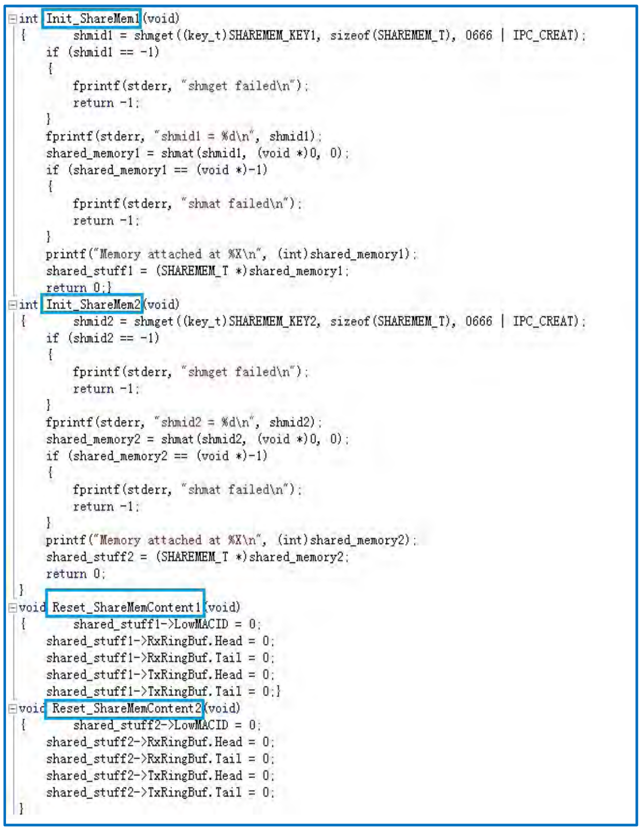
FIGURE 15. The signals used in Linux system.

only 31 can be used by the user thread including number 34 - number 64(i.e., highlighted in red). In our open HetNets platform, we use two of them including "(34) SIGRTMIN" and "(35) SIGRTMIN+1" as the virtual PHY thread communication ID.

Let us go back to the main function, code line 4 - code line 7 mainly focuses on allocating and initializing the share-memory. Fig.16 shows the definition of function "Init_ShareMem ()" and function "Reset_ShareMemContent ()". As we can see, in function "Init_ShareMem ()", the role of function "shmat()" is to start access to the shared memory and make the connection between the shared memory and the address space of virtual lower MAC thread. In function "Init_ShareMem ()", two transmission rings and two reception rings are respectively initialized. Then, the main function jump to code line 8.

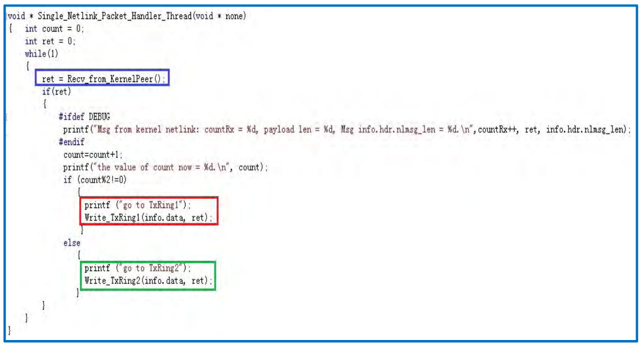
In code line 8, function "Set_LowMACID ()" use the obtained thread ID in code line 2 to set the virtual lower MAC thread ID. Then, in code line 9, Netlink socket is ready for startup and if it successfully initializes, the main function jumps to the key function "Single_Netlink_Packet_Handler_Thread ()" shown in code line 16. The function in code line 16 is a "bridge" (i.e., the pink arrow line in Fig.4) which connects the virtual upper MAC module and the virtual lower MAC module, and the detailed information of this function is shown in Fig.17.

In Fig.17, the "while (1)" statement and the function "Recv_from_KernelPeer ()" (i.e., the dark blue rectangle shown in Fig.17) demonstrates that the virtual lower MAC thread continuously receive the packets form the virtual upper MAC. The counter can start to add 1 once a packet arrives in virtual lower MAC, then based on the arrival packet's number, the virtual lower MAC (i.e., Offloading Engine) decide which path to select. For example, if the counter's value is an odd number, then virtual lower MAC selects path number two (i.e., the green rectangle shown in Fig.17) and writes this packet to TX2, otherwise, it selects path number one (i.e., the red rectangle shown in Fig.17) and writes this packet to TX1. The signal notification scheme is included in



```
int Init_ShareMem1(void)
{       shmid1 = shmget((key_t)SHAREMEM_KEY1, sizeof(SHAREMEM_T), 0666 | IPC_CREAT);
    if (shmid1 == -1)
    {
        fprintf(stderr, "shmget failed\n");
        return -1;
    }
    fprintf(stderr, "shmid1 = %d\n", shmid1);
    shared_memory1 = shmat(shmid1, (void *)0, 0);
    if (shared_memory1 == (void *)-1)
    {
        fprintf(stderr, "shmat failed\n");
        return -1;
    }
    printf("Memory attached at %X\n", (int)shared_memory1);
    shared_stuff1 = (SHAREMEM_T *)shared_memory1;
    return 0;}
int Init_ShareMem2(void)
{       shmid2 = shmget((key_t)SHAREMEM_KEY2, sizeof(SHAREMEM_T), 0666 | IPC_CREAT);
    if (shmid2 == -1)
    {
        fprintf(stderr, "shmget failed\n");
        return -1;
    }
    fprintf(stderr, "shmid2 = %d\n", shmid2);
    shared_memory2 = shmat(shmid2, (void *)0, 0);
    if (shared_memory2 == (void *)-1)
    {
        fprintf(stderr, "shmat failed\n");
        return -1;
    }
    printf("Memory attached at %X\n", (int)shared_memory2);
    shared_stuff2 = (SHAREMEM_T *)shared_memory2;
    return 0;
}
void Reset_ShareMemContent1(void)
{       shared_stuff1->LowMACID = 0;
    shared_stuff1->RxRingBuf.Head = 0;
    shared_stuff1->RxRingBuf.Tail = 0;
    shared_stuff1->TxRingBuf.Head = 0;
    shared_stuff1->TxRingBuf.Tail = 0;}
void Reset_ShareMemContent2(void)
{       shared_stuff2->LowMACID = 0;
    shared_stuff2->RxRingBuf.Head = 0;
    shared_stuff2->RxRingBuf.Tail = 0;
    shared_stuff2->TxRingBuf.Head = 0;
    shared_stuff2->TxRingBuf.Tail = 0;
}
```

FIGURE 16. The definition of function "Init_ShareMem ()" and function "Reset_ShareMemContent ()".

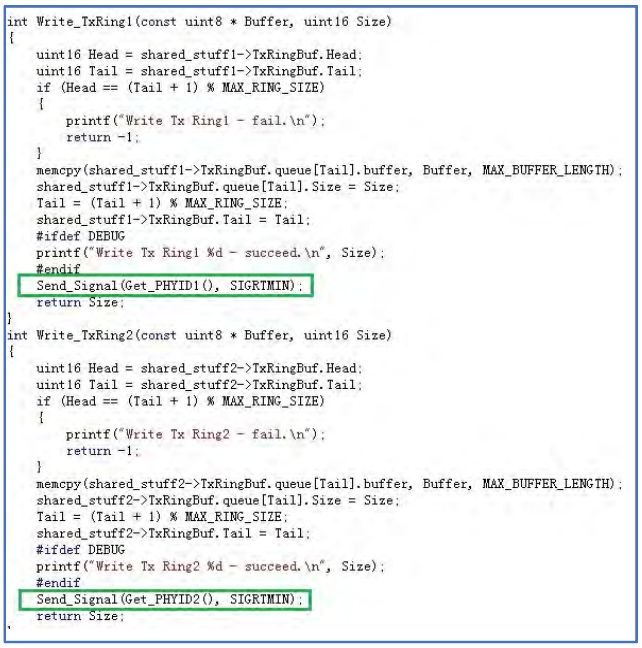


```
void * Single_Netlink_Packet_Handler_Thread(void * none)
{   int count = 0;
    int ret = 0;
    while(1)
    {
        ret = Recv_from_KernelPeer();
        if(ret)
        {
            #ifdef DEBUG
             printf("Msg from kernel netlink: countRx = %d, payload len = %d, Msg info.hdr.nlmsg_len = %d.\n",countRx++, ret, info.hdr.nlmsg_len);
            #endif
             count=count+1;
             printf("the value of count now = %d.\n", count);
             if (count%2!=0)
               {
                 printf ("go to TxRing1");
                 Write_TxRing1(info.data, ret);
               }
             else
               {
                 printf ("go to TxRing2");
                 Write_TxRing2(info.data, ret);
               }
        }
    }
}
```

FIGURE 17. Definition of function "Single_Netlink_Packet_Handler_Thread()".

function "Write_TxRing ()", and the definition of function "Write_TxRing ()" is depicted in Fig.18.

As shown in Fig.18, the share-memory use the ring structure to store the packets transmitted form virtual upper MAC. If the arrival packet is successfully put into the ring, then the virtual lower MAC thread utilizes function "Send_Signal ()" (i.e., the green rectangle shown in Fig.18) to notify (i.e., the red arrow lines in Fig.4) virtual PHY (1,2) thread it is the right time to read the TX (1,2). Also, we can see that the first formal parameter in function header is a function called "Get_PHYID ()", this function is used to obtain the virtual



```
int Write_TxRing1(const uint8 * Buffer, uint16 Size)
{
    uint16 Head = shared_stuff1->TxRingBuf.Head;
    uint16 Tail = shared_stuff1->TxRingBuf.Tail;
    if (Head == (Tail + 1) % MAX_RING_SIZE)
    {
        printf("Write Tx Ring1 - fail.\n");
        return -1;
    }
    memcpy(shared_stuff1->TxRingBuf.queue[Tail].buffer, Buffer, MAX_BUFFER_LENGTH);
    shared_stuff1->TxRingBuf.queue[Tail].Size = Size;
    Tail = (Tail + 1) % MAX_RING_SIZE;
    shared_stuff1->TxRingBuf.Tail = Tail;
    #ifdef DEBUG
    printf("Write Tx Ring1 %d - succeed.\n", Size);
    #endif
    Send_Signal(Get_PHYID1(), SIGRTMIN);
    return Size;
}
int Write_TxRing2(const uint8 * Buffer, uint16 Size)
{
    uint16 Head = shared_stuff2->TxRingBuf.Head;
    uint16 Tail = shared_stuff2->TxRingBuf.Tail;
    if (Head == (Tail + 1) % MAX_RING_SIZE)
    {
        printf("Write Tx Ring2 - fail.\n");
        return -1;
    }
    memcpy(shared_stuff2->TxRingBuf.queue[Tail].buffer, Buffer, MAX_BUFFER_LENGTH);
    shared_stuff2->TxRingBuf.queue[Tail].Size = Size;
    Tail = (Tail + 1) % MAX_RING_SIZE;
    shared_stuff2->TxRingBuf.Tail = Tail;
    #ifdef DEBUG
    printf("Write Tx Ring2 %d - succeed.\n", Size);
    #endif
    Send_Signal(Get_PHYID2(), SIGRTMIN);
    return Size;
}
```

FIGURE 18. The definition of function "Write_TxRing ()".

PHY thread ID (i.e., Offloading Observing Window 1 or 2). The second formal parameter in function "Get_PHYID ()" is a signal ID used by virtual lower MAC thread.

Till now, we have finished the analyzing process for the Offloading Engine module (i.e., virtual lower MAC module). In next section, we will focus on analyzing the Offloading Observing Window module (i.e., four purple rectangles shown in Fig.4) of the open HetNets platform.

D. OFFLOADING OBSERVING WINDOW MODULE

Fig.19 demonstrates the function caller diagram in Offloading Observing Window module (i.e., virtual PHY module). Like virtual lower MAC module, the virtual PHY module also calls other six functions when in startup status. As previously mentioned, code files "signalhandler.h" and "signalhandler.c" are used to conduct the signal notification mechanism between the virtual lower MAC module and the virtual PHY module, and code files "sharemem.h" and "sharemem.c" are used to process the share-memory mechanism between them. Different from the virtual lower MAC module, the code files "UDP_Trxer.h" and "UDP_Trxer.c" in virtual PHY module are responsible for the communication between the remote UE's virtual PHY thread and the local virtual PHY thread. To be specific, based on Fig.4, the communication path between the remote UE's virtual PHY thread and the local virtual PHY thread is "open HetNets platform's virtual PHY thread – port1/port2 – open WiFi platform /open LTE platform – UE's virtual PHY thread. To illustrate this module's logical function more clearly, in the next, we give the detailed descriptions for each code file. Here, we make some further explanations for the virtual PHY thread. As shown in Fig.4, those two virtual PHY threads are two concurrent

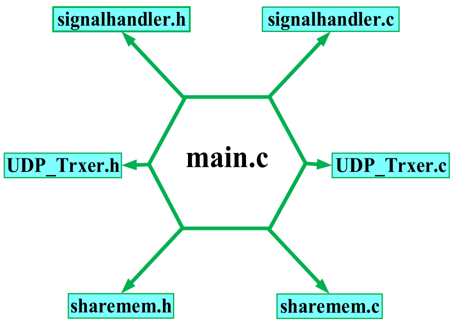
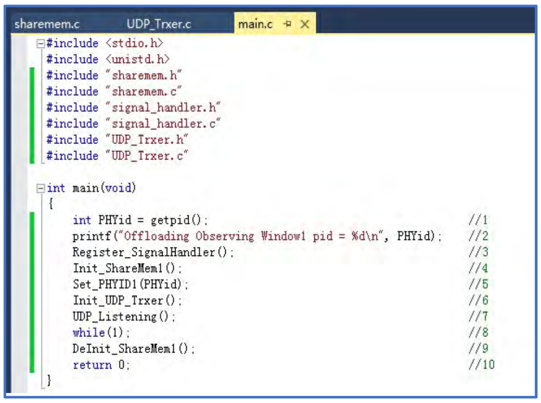


FIGURE 19. The function caller diagram in Offloading Observing Window module.



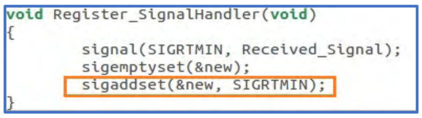
```
#include <stdio.h>
#include <unistd.h>
#include "sharemem.h"
#include "sharemem.c"
#include "signal_handler.h"
#include "signal_handler.c"
#include "UDP_Trxer.h"
#include "UDP_Trxer.c"

int main(void)
{
    int PHYid = getpid();                                              //1
    printf("Offloading Observing Window1 pid = %d\n", PHYid);          //2
    Register_SignalHandler();                                          //3
    Init_ShareMem1();                                                  //4
    Set_PHYID1(PHYid);                                                 //5
    Init_UDP_Trxer();                                                  //6
    UDP_Listening();                                                   //7
    while(1);                                                          //8
    DeInit_ShareMem1();                                                //9
    return 0;                                                          //10
}
```

FIGURE 20. The main function in Offloading Observing Window module.

UDP threads, and each of them follows a one-to-one match with the registered signal shown in Fig.15 (i.e., those highlighted signals that could be used by user thread). Therefore, the maximum possible number of threads allowed to run concurrently in our open HetNets platform is 31 (i.e., signal (ID 34)-signal (ID 64) in Fig.15). In section III.E, we will exploit the concurrent mechanism to improve the throughput of the open HetNets platform.

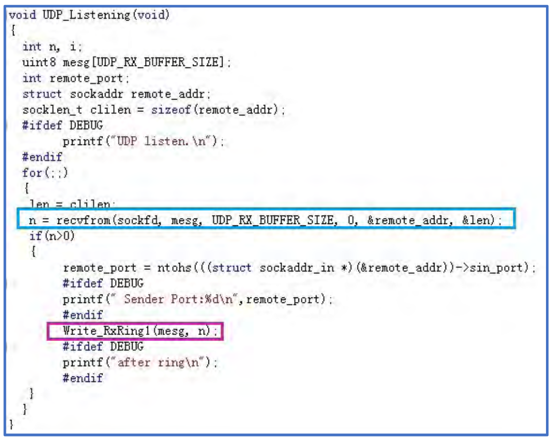
The main function of virtual PHY module is shown in Fig.20, as illustrated in code line 1, function "getpid ()" is used to get the virtual PHY thread ID (i.e., PHYid). Unlike virtual lower MAC module, in code line 3, function "Register_SignalHandler ()" register only one notification signal (i.e., the received signal transmit from virtual lower MAC thread) used by virtual PHY thread with the reason that there is only one virtual lower MAC module. The definition of "Register_SignalHandler ()" is included in code file "signalhandler.c" and it is shown in Fig.21.



```
void Register_SignalHandler(void)
{
        signal(SIGRTMIN, Received_Signal);
        sigemptyset(&new);
        sigaddset(&new, SIGRTMIN);
}
```

FIGURE 21. The definition of function "Register_SignalHandler ()" in virtual PHY module.

According to Fig.21, we can observe that the function "Register_SignalHandler ()" defined in virtual PHY module is the same with that defined in virtual lower MAC module. Specifically, in code line 1, function "signal ()" first register the signal SIGRTMIN as the received signal of virtual PHY thread. Then, the function "sigemptyset ()" empty the original signal set in virtual PHY thread and initialize the new registered signal. Finally, function "sigaddset ()" add the new registered signal to the signal set of virtual PHY thread. Based on above mentioned processes, the main function jump to code line 4, function "Init_ShareMem()" in code line 4 mainly responsible for allocating and initializing the share-memory and this function is same as that define in virtual lower MAC module. As the main function continues to execute, function "Set_PHYID ()" in code line 5 uses the obtained thread ID to set the virtual PHY ID. Then, the main function calls function "Init_UDP_Trxer" shown in code line 6, and the UDP TX & RX threads are ready for startup. If the UDP TX & RX threads are successfully initialize, the main function jumps to the key function "UDP_Listening ()" shown in code line 7. The function in code line 7 is a "peer-to-peer bridge" which can make connection between the UDP thread located in the open HetNets platform and the UDP thread located in UE. The detailed information of function "UDP_Listening ()" is shown in Fig.22.

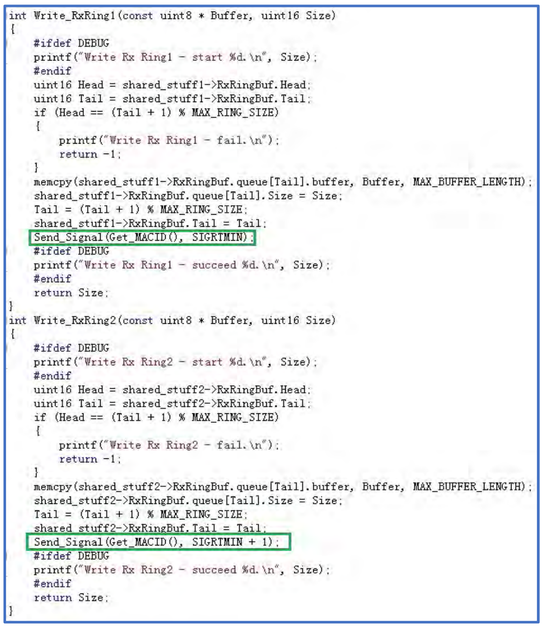


```
void UDP_Listening(void)
{
  int n, i;
  uint8 mesg[UDP_RX_BUFFER_SIZE];
  int remote_port;
  struct sockaddr remote_addr;
  socklen_t clilen = sizeof(remote_addr);
  #ifdef DEBUG
        printf("UDP listen.\n");
  #endif
  for(;;)
  {
   len = clilen;
   n = recvfrom(sockfd, mesg, UDP_RX_BUFFER_SIZE, 0, &remote_addr, &len);
   if(n>0)
   {
        remote_port = ntohs(((struct sockaddr_in *)(&remote_addr))->sin_port);
        #ifdef DEBUG
        printf(" Sender Port:%d\n",remote_port);
        #endif
        Write_RxRing1(mesg, n);
        #ifdef DEBUG
        printf("after ring\n");
        #endif
   }
  }
}
```

FIGURE 22. The definition of function "UDP_Listening ()".

In Fig.22, the "for" statement and the function "recvfrom ()" in light blue rectangle demonstrates that the virtual PHY thread continuously receive the packets from its peer side by port (1, 2). If the received byte number larger than 0, then virtual PHY thread writes the arrival packets to RX(1, 2) ring and send signal to virtual lower MAC thread to notify it is the right time to read the RX(1, 2) ring. The signal notification scheme is included in function "Write_RxRing ()", and the definition of "Write_RxRing ()" is demonstrated in Fig.23.

As shown in Fig.23, the share-memory use the ring structure to store the packets transmitted form UE side. If the arrival packet is successfully put into the RX ring, then the virtual PHY thread utilizes function "Send_Signal ()" (i.e., the green rectangle shown in Fig.23) to notify (i.e., the dark



```
int Write_RxRing1(const uint8 * Buffer, uint16 Size)
{
    #ifdef DEBUG
    printf("Write Rx Ring1 - start %d.\n", Size);
    #endif
    uint16 Head = shared_stuff1->RxRingBuf.Head;
    uint16 Tail = shared_stuff1->RxRingBuf.Tail;
    if (Head == (Tail + 1) % MAX_RING_SIZE)
     {
        printf("Write Rx Ring1 - fail.\n");
        return -1;
     }
    memcpy(shared_stuff1->RxRingBuf.queue[Tail].buffer, Buffer, MAX_BUFFER_LENGTH);
    shared_stuff1->RxRingBuf.queue[Tail].Size = Size;
    Tail = (Tail + 1) % MAX_RING_SIZE;
    shared_stuff1->RxRingBuf.Tail = Tail;
    Send_Signal(Get_MACID(), SIGRTMIN);
    #ifdef DEBUG
    printf("Write Rx Ring1 - succeed %d.\n", Size);
    #endif
    return Size;
}
int Write_RxRing2(const uint8 * Buffer, uint16 Size)
{
    #ifdef DEBUG
    printf("Write Rx Ring2 - start %d.\n", Size);
    #endif
    uint16 Head = shared_stuff2->RxRingBuf.Head;
    uint16 Tail = shared_stuff2->RxRingBuf.Tail;
    if (Head == (Tail + 1) % MAX_RING_SIZE)
     {
        printf("Write Rx Ring2 - fail.\n");
        return -1;
     }
    memcpy(shared_stuff2->RxRingBuf.queue[Tail].buffer, Buffer, MAX_BUFFER_LENGTH);
    shared_stuff2->RxRingBuf.queue[Tail].Size = Size;
    Tail = (Tail + 1) % MAX_RING_SIZE;
    shared_stuff2->RxRingBuf.Tail = Tail;
    Send_Signal(Get_MACID(), SIGRTMIN + 1);
    #ifdef DEBUG
    printf("Write Rx Ring2 - succeed %d.\n", Size);
    #endif
    return Size;
}
```

FIGURE 23. The definition of function "Write_RxRing ()".

blue arrow lines in Fig.4) virtual lower MAC thread it is the right time to read the RX ring (1,2). Also, we can see that the first formal parameter in function header is a function called "Get_MACID ()", this function is used to obtain the virtual lower MAC thread ID (i.e., Offloading Engine). The second formal parameter in function "Get_PHYID ()" is a signal ID used by virtual PHY thread.

Till now, we have finished the analyzing process for the Offloading Observing Window module (i.e., virtual PHY module). In next section, we will first run the HetNets platform to see whether it can work correctly and then do some optimizations for it.

E. SOFTWARE DEFINED OPEN HETNETS PLATFORM OPTIMIZATION

Based on the above-mentioned analysis, in this section, we first conduct the packet transmission experiment to check out whether the open HetNets platform can work correctly. Then, we do some optimizations for the HetNets platform's software architecture to improve its overall throughput.

To be specific, the packet transmission experiment plan is shown in Fig.24, we deploy 3 software-defined devices including the software-defined open WiFi platform (i.e., the yellow bidirectional arrow line in Fig.4) [6], the software-defined open LTE platform (i.e., the light green bidirectional arrow line in Fig.4) [7], and the software-defined user equipment (UE) in our laboratory. As illustrated in Fig.25, each software-defined device is composed by the USRP B210 and the INTEL NUL. The reason for choosing INTEL NUC is that it is a high-performance hardware platform with small-size appearance (i.e., 21 Centimeters in length and 11 centimeters in width), which cannot occupy much area and the reason for choosing the USRP as the RF front-end is that it is

TABLE 1. The experiment parameters.

Experiment Devices	Hardware Platform	Software Platform
3 INTEL NUC 3 USRP B210 1 Host PC	**INTEL NUC:** Memory: 31.3 GB Processor: Intel® Core™ i7-6770HQ CPU @ 2.60GHz OS type: 64-bit Disk: 470.2 GB Appearance: 21 Centimeters in length and 11 centimeters in width **USRP-B210:** continuous RF coverage from 70 MHz – 6 GHz Full duplex, MIMO (2 Tx & 2 Rx) operation with up to 56 MHz of real-time bandwidth (61.44MS/s quadrature) Fast and convenient SuperSpeed USB 3.0 connectivity 50 MS/s Rx 45 MS/s Tx (Max flexible) **Host PC:** RAM: 16.0 GB (15.9 GB available) Processor: Intel(R) Core (TM)i5-6500CPU@3.20 GHz OS type: 64 bits Hard disk: 1200 GB	**OS Version:** Ubuntu 14.04.5 **OS Core:** 4.7.1 **Peripheral Device Driver:** open-source USRP Hardwar Driver™ (UHD) **Linux Application:** Software-defined Open WiFi platform Software-defined Open LTE platform Software-defined UE platform

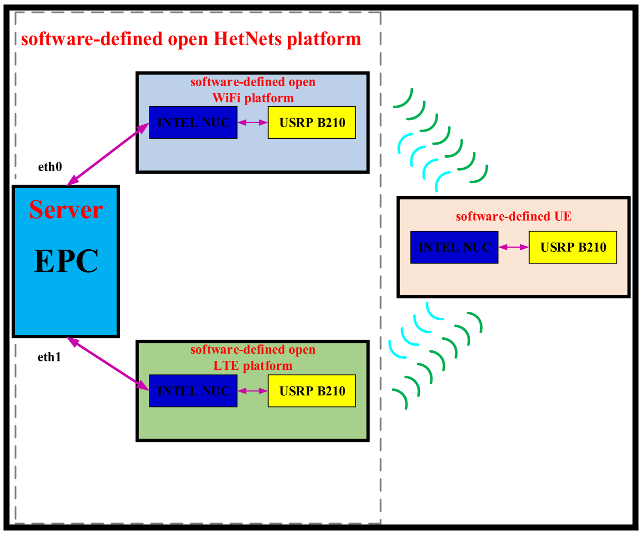

software-defined open HetNets platform
software-defined open WiFi platform
INTEL NUC
USRP B210
eth0
Server
EPC
software-defined UE
INTEL NUC
USRP B210
eth1
software-defined open LTE platform
INTEL NUC
USRP B210

FIGURE 24. The experiment plan which is corresponding to the architecture shown in Fig.4.

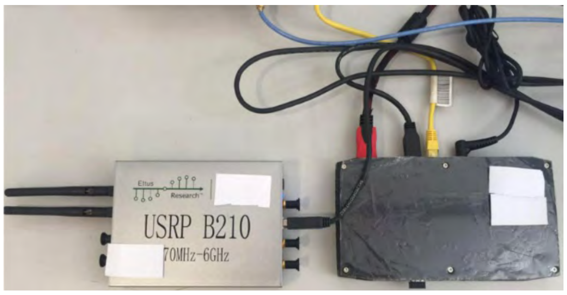


FIGURE 25. The software-defined device composed by USRP-B210 and INTEL NUC.

a high-performance and repeated programming device. Let the EPC locate on a server with two ethernet interfaces (i.e., eth0 and eth1), one interface connects to the software-defined open WiFi platform and the other connects to the software defined open LTE platform. Based on the experiment plan, we then give the detailed experiment parameters shown in Table 1.

Based on the above-mentioned experiment plan and experiment parameters table, we carry out the following experiment procedures:

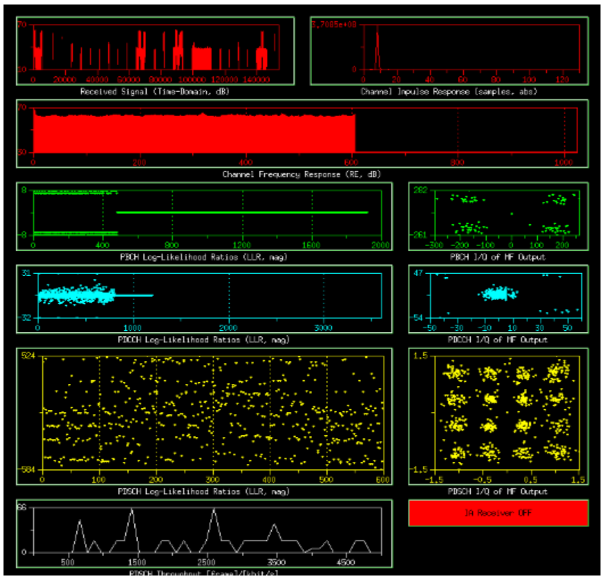

Received Signal (Time-Domain, dB)
Channel Impulse Response (samples, abs)
Channel Frequency Response (RE, dB)
PBCH Log-Likelihood Ratios (LLR, mag)
PBCH I/Q of MF Output
PDCCH Log-Likelihood Ratios (LLR, mag)
PDCCH I/Q of MF Output
PDSCH Log-Likelihood Ratios (LLR, mag)
PDSCH I/Q of MF Output
IA Receiver OFF
PDSCH Throughput [frame]/[kbit/s]

FIGURE 26. The shooting screen of physical layer statistic information shown in software defined UE when it connects to the software defined open HetNets platform.

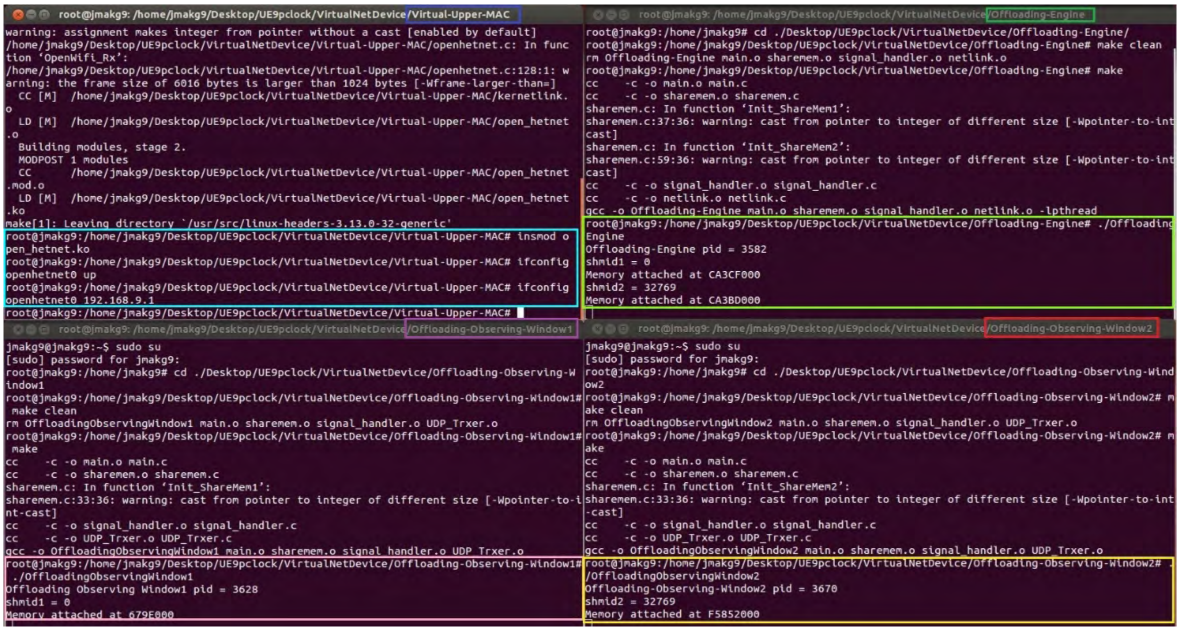
(1). We first implement the Virtual Upper MAC module, the Offloading Engine module, and the Offloading Observing Window module into the server.

(2). We then compile and run the software-defined open WiFi platform, the software-defined open LTE platform, and the software-defined UE.

(3). After those three software-defined devices successfully startup, we connect soft UE to another two devices (i.e., software-defined open WiFi platform and software-defined open LTE platform). Then, in UE side, we can see the physical layer statistic information shown in Fig.26 appears, and it is demonstrating that the soft UE has connected to the software defined open HetNets platform.

(4). Then, in server side, we compile and initialize the Virtual Upper MAC module, the Offloading Engine module, and the Offloading Observing Window module. The detailed processes are shown from Fig.27-Fig.28.

As shown in Fig.27, the little dark blue rectangle highlights the Virtual Upper MAC window, in this window, we first



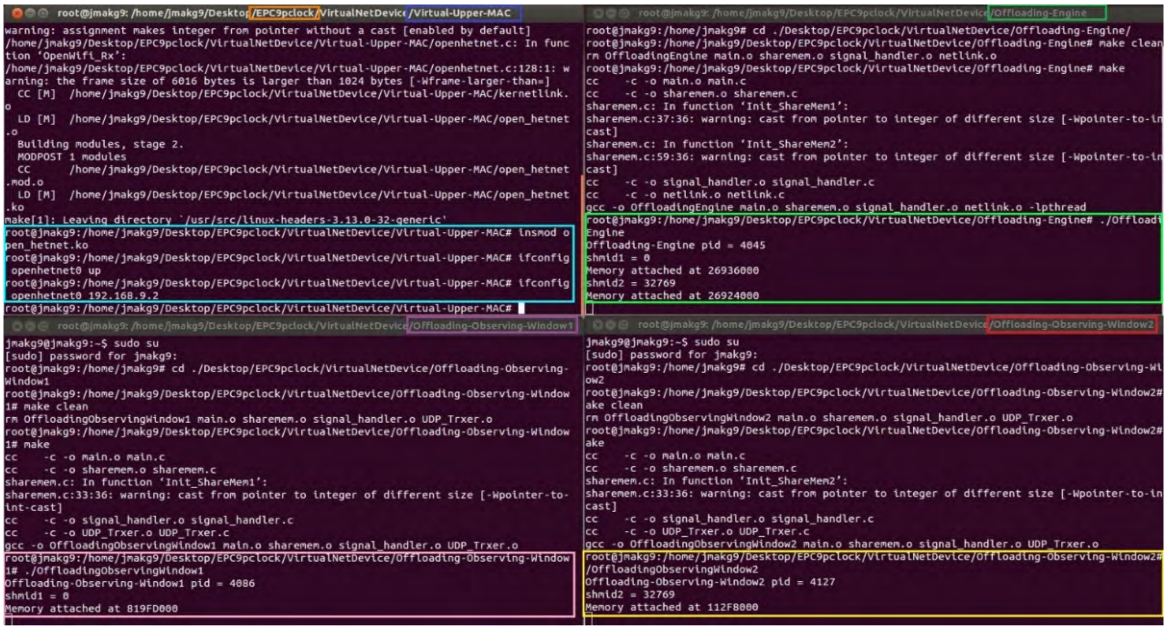
```
root@jmakg9: /home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC
warning: assignment makes integer from pointer without a cast [enabled by default]
/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC/openhetnet.c: In func
tion 'OpenWifi_Rx':
/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC/openhetnet.c:128:1: w
arning: the frame size of 6016 bytes is larger than 1024 bytes [-Wframe-larger-than=]
  CC [M]  /home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC/kernetlink.
o
  LD [M]  /home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC/open_hetnet
.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC/open_hetnet
.mod.o
  LD [M]  /home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC/open_hetnet
.ko
make[1]: Leaving directory `/usr/src/linux-headers-3.13.0-32-generic'
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC# insmod o
pen_hetnet.ko
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC# ifconfig
openhetnet0 up
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC# ifconfig
openhetnet0 192.168.9.1
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC#
```

```
root@jmakg9: /home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Offloading-Engine
root@jmakg9:/home/jmakg9# cd ./Desktop/UE9pclock/VirtualNetDevice/Offloading-Engine/
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Offloading-Engine# make clean
rm Offloading-Engine main.o sharemem.o signal_handler.o netlink.o
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Offloading-Engine# make
cc    -c -o main.o main.c
cc    -c -o sharemem.o sharemem.c
sharemem.c: In function 'Init_ShareMem1':
sharemem.c:37:36: warning: cast from pointer to integer of different size [-Wpointer-to-int
cast]
sharemem.c: In function 'Init_ShareMem2':
sharemem.c:59:36: warning: cast from pointer to integer of different size [-Wpointer-to-int
cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o netlink.o netlink.c
gcc -o Offloading-Engine main.o sharemem.o signal_handler.o netlink.o -lpthread
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Offloading-Engine# ./Offloading
Engine
Offloading-Engine pid = 3582
shmid1 = 0
Memory attached at CA3CF000
shmid2 = 32769
Memory attached at CA3BD000
```

```
root@jmakg9: /home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Offloading-Observing-Window1
jmakg9@jmakg9:~$ sudo su
[sudo] password for jmakg9:
root@jmakg9:/home/jmakg9# cd ./Desktop/UE9pclock/VirtualNetDevice/Offloading-Observing-W
indow1
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Offloading-Observing-Window1#
 make clean
rm OffloadingObservingWindow1 main.o sharemem.o signal_handler.o UDP_Trxer.o
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Offloading-Observing-Window1#
 make
cc    -c -o main.o main.c
cc    -c -o sharemem.o sharemem.c
sharemem.c: In function 'Init_ShareMem1':
sharemem.c:33:36: warning: cast from pointer to integer of different size [-Wpointer-to-i
nt-cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow1 main.o sharemem.o signal_handler.o UDP_Trxer.o
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Offloading-Observing-Window1#
 ./OffloadingObservingWindow1
Offloading Observing Window1 pid = 3628
shmid1 = 0
Memory attached at 679E000
```

```
root@jmakg9: /home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Offloading-Observing-Window2
jmakg9@jmakg9:~$ sudo su
[sudo] password for jmakg9:
root@jmakg9:/home/jmakg9# cd ./Desktop/UE9pclock/VirtualNetDevice/Offloading-Observing-Wind
ow2
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Offloading-Observing-Window2# m
ake clean
rm OffloadingObservingWindow2 main.o sharemem.o signal_handler.o UDP_Trxer.o
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Offloading-Observing-Window2# m
ake
cc    -c -o main.o main.c
cc    -c -o sharemem.o sharemem.c
sharemem.c: In function 'Init_ShareMem2':
sharemem.c:33:36: warning: cast from pointer to integer of different size [-Wpointer-to-int
-cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow2 main.o sharemem.o signal_handler.o UDP_Trxer.o
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Offloading-Observing-Window2# .
/OffloadingObservingWindow2
Offloading-Observing-Window2 pid = 3670
shmid2 = 32769
Memory attached at F5852000
```

FIGURE 27. The shooting screens related to the initialization information of Virtual-Upper-MAC, Offloading-Engine, Offloading-Observing-Window1, and Offloading-Observing-Window2 in software defined UE.



```
root@jmakg9: /home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC
warning: assignment makes integer from pointer without a cast [enabled by default]
/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC/openhetnet.c: In func
tion 'OpenWifi_Rx':
/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC/openhetnet.c:128:1: w
arning: the frame size of 6016 bytes is larger than 1024 bytes [-Wframe-larger-than=]
  CC [M]  /home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC/kernetlink.
o
  LD [M]  /home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC/open_hetnet
.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC/open_hetnet
.mod.o
  LD [M]  /home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC/open_hetnet
.ko
make[1]: Leaving directory `/usr/src/linux-headers-3.13.0-32-generic'
root@jmakg9:/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC# insmod o
pen_hetnet.ko
root@jmakg9:/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC# ifconfig
 openhetnet0 up
root@jmakg9:/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC# ifconfig
 openhetnet0 192.168.9.2
root@jmakg9:/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC#
```

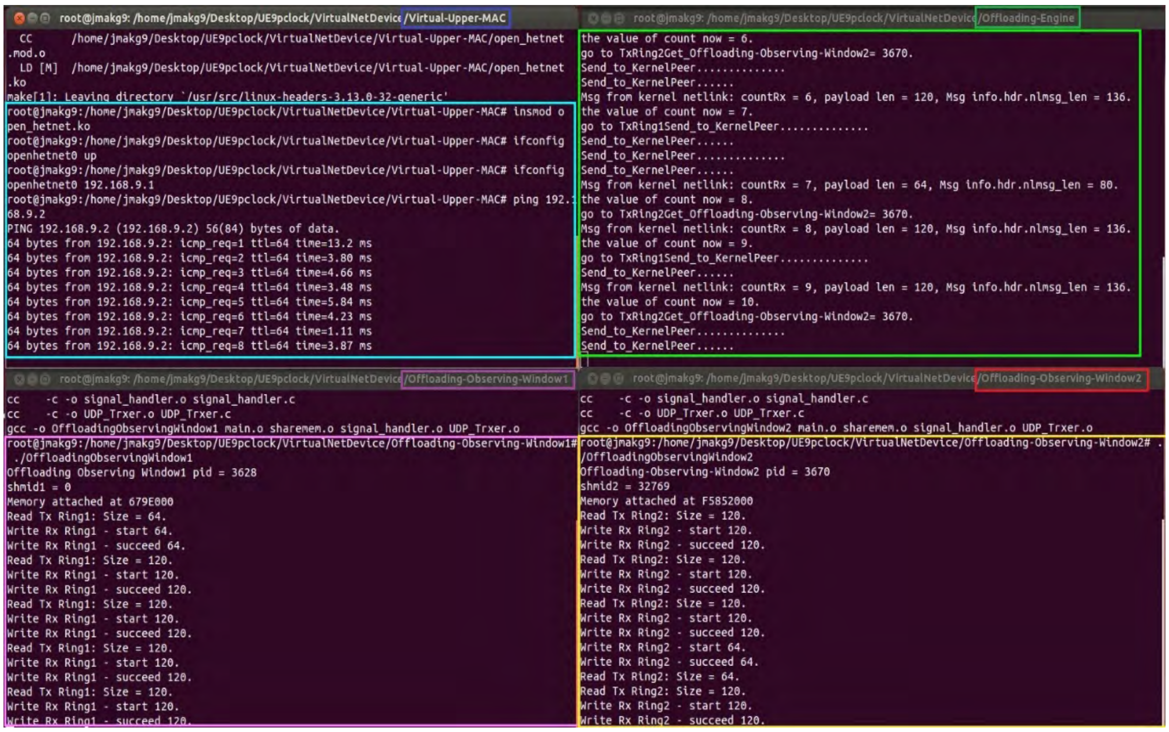
```
root@jmakg9: /home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Offloading-Engine
root@jmakg9:/home/jmakg9# cd ./Desktop/EPC9pclock/VirtualNetDevice/Offloading-Engine/
root@jmakg9:/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Offloading-Engine# make clean
rm OffloadingEngine main.o sharemem.o signal_handler.o netlink.o
root@jmakg9:/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Offloading-Engine# make
cc    -c -o main.o main.c
cc    -c -o sharemem.o sharemem.c
sharemem.c: In function 'Init_ShareMem1':
sharemem.c:37:36: warning: cast from pointer to integer of different size [-Wpointer-to-in
cast]
sharemem.c: In function 'Init_ShareMem2':
sharemem.c:59:36: warning: cast from pointer to integer of different size [-Wpointer-to-in
cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o netlink.o netlink.c
gcc -o OffloadingEngine main.o sharemem.o signal_handler.o netlink.o -lpthread
root@jmakg9:/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Offloading-Engine# ./Offload
Engine
Offloading-Engine pid = 4045
shmid1 = 0
Memory attached at 26936000
shmid2 = 32769
Memory attached at 26924000
```

```
root@jmakg9: /home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Offloading-Observing-Window1
jmakg9@jmakg9:~$ sudo su
[sudo] password for jmakg9:
root@jmakg9:/home/jmakg9# cd ./Desktop/EPC9pclock/VirtualNetDevice/Offloading-Observing-
Window1
root@jmakg9:/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Offloading-Observing-Window
1# make clean
rm OffloadingObservingWindow1 main.o sharemem.o signal_handler.o UDP_Trxer.o
root@jmakg9:/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Offloading-Observing-Window
1# make
cc    -c -o main.o main.c
cc    -c -o sharemem.o sharemem.c
sharemem.c: In function 'Init_ShareMem1':
sharemem.c:33:36: warning: cast from pointer to integer of different size [-Wpointer-to-
int-cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow1 main.o sharemem.o signal_handler.o UDP_Trxer.o
root@jmakg9:/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Offloading-Observing-Window
1# ./OffloadingObservingWindow1
Offloading-Observing-Window1 pid = 4086
shmid1 = 0
Memory attached at 819FD000
```

```
root@jmakg9: /home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Offloading-Observing-Window2
jmakg9@jmakg9:~$ sudo su
[sudo] password for jmakg9:
root@jmakg9:/home/jmakg9# cd ./Desktop/EPC9pclock/VirtualNetDevice/Offloading-Observing-Wi
ow2
root@jmakg9:/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Offloading-Observing-Window2#
ake clean
rm OffloadingObservingWindow2 main.o sharemem.o signal_handler.o UDP_Trxer.o
root@jmakg9:/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Offloading-Observing-Window2#
ake
cc    -c -o main.o main.c
cc    -c -o sharemem.o sharemem.c
sharemem.c: In function 'Init_ShareMem2':
sharemem.c:33:36: warning: cast from pointer to integer of different size [-Wpointer-to-in
cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow2 main.o sharemem.o signal_handler.o UDP_Trxer.o
root@jmakg9:/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Offloading-Observing-Window2#
/OffloadingObservingWindow2
Offloading-Observing-Window2 pid = 4127
shmid2 = 32769
Memory attached at 112F8000
```

FIGURE 28. The shooting screens related to the initialization information of Virtual-Upper-MAC, Offloading-Engine, Offloading-Observing-Window1, and Offloading-Observing-Window2 in software defined open HetNets platform.

input the commands "make clean" and "make" to compile Virtual Upper MAC module. Then we input commands "insmod open_hetnet.ko", "ifconfig openhetnet0 up", and "ifconfig openhetnet0 192.168.9.1" to insert the virtual lower MAC module to OS kernel and configure the ip address of virtual interface "openhetnet0" to "192.168.9.1" (i.e., the information highlighted in bigger light blue rectangle). The little dark green rectangle highlights the Offloading Engine window, in this window, we first input the commands "make clean" and "make" to compile Offloading Engine module (i.e., virtual lower MAC module), and then we input commands "./Offloading Engine" to initialize it (i.e., the initialization information highlighted in bigger light green rectangle). The little purple rectangle and red rectangle highlight those two Offloading Observing Window modules (i.e., two virtual PHY modules), in those two windows, we first input



root@jmakg9: /home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC

```
  CC      /home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC/open_hetnet
.mod.o
  LD [M]  /home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC/open_hetnet
.ko
make[1]: Leaving directory `/usr/src/linux-headers-3.13.0-32-generic'
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC# insmod o
pen_hetnet.ko
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC# ifconfig
openhetnet0 up
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC# ifconfig
openhetnet0 192.168.9.1
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Virtual-Upper-MAC# ping 192.1
68.9.2
PING 192.168.9.2 (192.168.9.2) 56(84) bytes of data.
64 bytes from 192.168.9.2: icmp_req=1 ttl=64 time=13.2 ms
64 bytes from 192.168.9.2: icmp_req=2 ttl=64 time=3.80 ms
64 bytes from 192.168.9.2: icmp_req=3 ttl=64 time=4.66 ms
64 bytes from 192.168.9.2: icmp_req=4 ttl=64 time=3.48 ms
64 bytes from 192.168.9.2: icmp_req=5 ttl=64 time=5.84 ms
64 bytes from 192.168.9.2: icmp_req=6 ttl=64 time=4.23 ms
64 bytes from 192.168.9.2: icmp_req=7 ttl=64 time=1.11 ms
64 bytes from 192.168.9.2: icmp_req=8 ttl=64 time=3.87 ms
```

root@jmakg9: /home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Offloading-Engine

```
the value of count now = 6.
go to TxRing2Get_Offloading-Observing-Window2= 3670.
Send_to_KernelPeer.............
Send_to_KernelPeer......
Msg from kernel netlink: countRx = 6, payload len = 120, Msg info.hdr.nlmsg_len = 136.
the value of count now = 7.
go to TxRing1Send_to_KernelPeer.............
Send_to_KernelPeer......
Send_to_KernelPeer.............
Send_to_KernelPeer......
Msg from kernel netlink: countRx = 7, payload len = 64, Msg info.hdr.nlmsg_len = 80.
the value of count now = 8.
go to TxRing2Get_Offloading-Observing-Window2= 3670.
Msg from kernel netlink: countRx = 8, payload len = 120, Msg info.hdr.nlmsg_len = 136.
the value of count now = 9.
go to TxRing1Send_to_KernelPeer.............
Send_to_KernelPeer......
Msg from kernel netlink: countRx = 9, payload len = 120, Msg info.hdr.nlmsg_len = 136.
the value of count now = 10.
go to TxRing2Get_Offloading-Observing-Window2= 3670.
Send_to_KernelPeer.............
Send_to_KernelPeer......
```

root@jmakg9: /home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Offloading-Observing-Window1

```
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow1 main.o sharemem.o signal_handler.o UDP_Trxer.o
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Offloading-Observing-Window1#
 ./OffloadingObservingWindow1
Offloading Observing Window1 pid = 3628
shmid1 = 0
Memory attached at 679E000
Read Tx Ring1: Size = 64.
Write Rx Ring1 - start 64.
Write Rx Ring1 - succeed 64.
Read Tx Ring1: Size = 120.
Write Rx Ring1 - start 120.
Write Rx Ring1 - succeed 120.
Read Tx Ring1: Size = 120.
Write Rx Ring1 - start 120.
Write Rx Ring1 - succeed 120.
Read Tx Ring1: Size = 120.
Write Rx Ring1 - start 120.
Write Rx Ring1 - succeed 120.
Read Tx Ring1: Size = 120.
Write Rx Ring1 - start 120.
Write Rx Ring1 - succeed 120.
```

root@jmakg9: /home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Offloading-Observing-Window2

```
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow2 main.o sharemem.o signal_handler.o UDP_Trxer.o
root@jmakg9:/home/jmakg9/Desktop/UE9pclock/VirtualNetDevice/Offloading-Observing-Window2# .
/OffloadingObservingWindow2
Offloading-Observing-Window2 pid = 3670
shmid2 = 32769
Memory attached at F5852000
Read Tx Ring2: Size = 120.
Write Rx Ring2 - start 120.
Write Rx Ring2 - succeed 120.
Read Tx Ring2: Size = 120.
Write Rx Ring2 - start 120.
Write Rx Ring2 - succeed 120.
Read Tx Ring2: Size = 120.
Write Rx Ring2 - start 120.
Write Rx Ring2 - succeed 120.
Write Rx Ring2 - start 64.
Write Rx Ring2 - succeed 64.
Read Tx Ring2: Size = 64.
Read Tx Ring2: Size = 120.
Write Rx Ring2 - start 120.
Write Rx Ring2 - succeed 120.
```

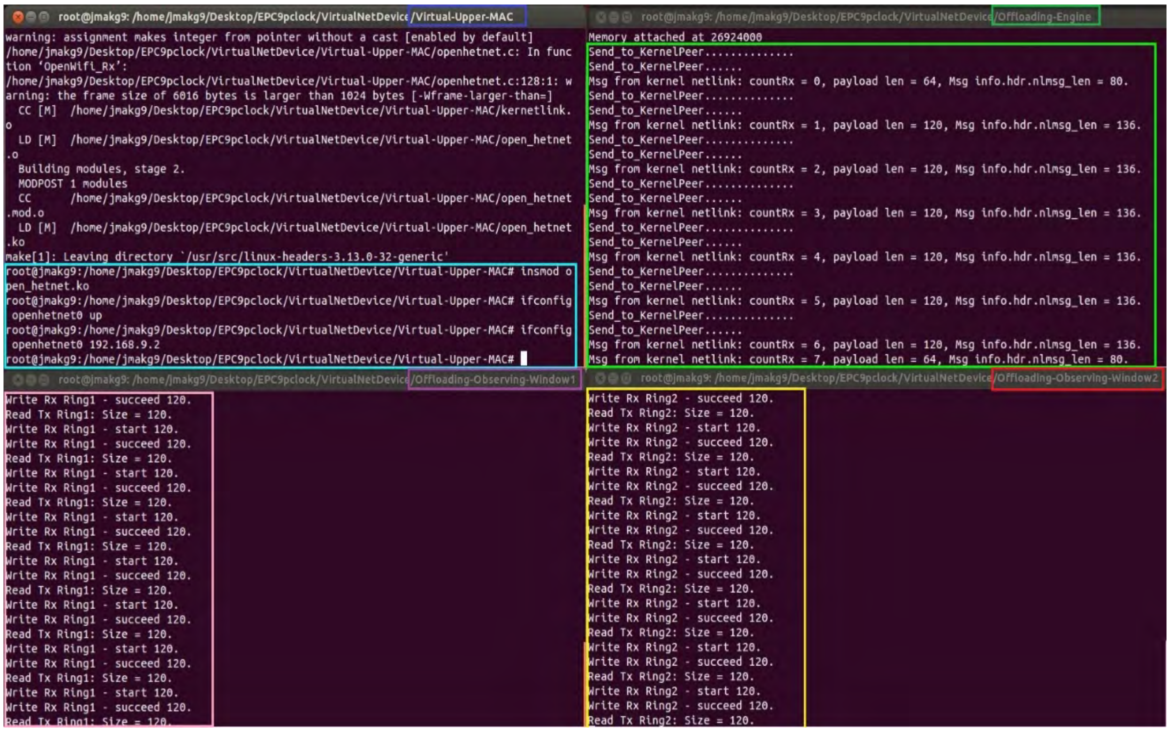
FIGURE 29. **The shooting screens related to the startup information of Virtual-Upper-MAC, Offloading-Engine, Offloading-Observing-Window1, and Offloading-Observing-Window2 in software defined UE.**

the commands "make clean" and "make" to compile them and then we input commands "./Offloading Observing Window1" and "./Offloading Observing Window2" to initialize them (i.e., the initialization information highlighted in pink rectangle and yellow rectangle).

In the open HetNets platform, we conduct the same initialization works as that in UE side. As shown in Fig.28, there are four shooting screens, which are respectively the Virtual Upper MAC window highlighted by little dark blue rectangle, Offloading Engine window highlighted by little dark green rectangle, and two Offloading Observing Window highlighted by purple rectangle and red rectangle. The command in the bigger light blue rectangle demonstrates that the ip address of the virtual net device in the HetNets platform has been configured as "192.168.99.2". Another three rectangles including the bigger green rectangle, pink rectangle, and yellow rectangle have demonstrated the HetNets platform has been successfully initialized.

(5). Based on above-mentioned procedures, in UE side, we input command "ping 192.168.9.2" to send packets to HetNets platform. Then, four screens appear in UE, which are the Virtual Upper MAC startup screen, Offloading Engine startup screen, and two Offloading Observing Window startup screens. The detailed information is shown in Fig.29, note that the little dark blue rectangle indicates the Virtual Upper MAC startup screen, the little dark green rectangle indicates the Offloading Engine startup screen, the little purple rectangle indicates the first Offloading Observing Window startup screen, and the little red rectangle indicates the second Offloading Observing Window startup screen. The bigger light blue rectangle shows that the sending packets have passed the Virtual Upper MAC and gone to the "Offloading Engine". After then, the messages "Read Tx Ring1" and "Read Tx Ring2" appear in the pink rectangle and yellow rectangle have demonstrated that those two virtual PHY threads have received the packets transmitted from virtual lower MAC (i.e., Offloading Engine) and have sent those packets to HetNets platform. Moreover, the messages "Write Rx Ring1" and "Write Rx Ring2" appear in the pink rectangle and yellow rectangle also demonstrated that those two virtual PHY threads have received the reply packets from HetNets platform.

Following Fig.29, another four screens appear in HetNets platform, which are the Virtual Upper MAC startup screen, Offloading Engine startup screen, and two Offloading Observing Window startup screens. The detailed information is shown in Fig.30, note that the little dark blue rectangle indicates the Virtual Upper MAC startup screen, the little dark green rectangle indicates the Offloading Engine startup screen, the little purple rectangle indicates the first Offloading Observing Window startup screen, and the little red rectangle indicates the second Offloading Observing Window startup screen. Moreover, the bigger light green rectangle, the pink rectangle, and the yellow rectangle demonstrate the HetNets platform has received the packets transmitted from UE (i.e., "Write Rx Ring1" in pink rectangle and "Write



root@jmakg9: /home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC

```
warning: assignment makes integer from pointer without a cast [enabled by default]
/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC/openhetnet.c: In func
tion 'OpenWifi_Rx':
/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC/openhetnet.c:128:1: w
arning: the frame size of 6016 bytes is larger than 1024 bytes [-Wframe-larger-than=]
  CC [M]  /home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC/kernetlink.
o
  LD [M]  /home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC/open_hetnet
.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC/open_hetnet
.mod.o
  LD [M]  /home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC/open_hetnet
.ko
make[1]: Leaving directory `/usr/src/linux-headers-3.13.0-32-generic'
root@jmakg9:/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC# insmod o
pen_hetnet.ko
root@jmakg9:/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC# ifconfig
 openhetnet0 up
root@jmakg9:/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC# ifconfig
 openhetnet0 192.168.9.2
root@jmakg9:/home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Virtual-Upper-MAC# 
```

root@jmakg9: /home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Offloading-Engine

```
Memory attached at 26924000
Send_to_KernelPeer.............
Send_to_KernelPeer......
Msg from kernel netlink: countRx = 0, payload len = 64, Msg info.hdr.nlmsg_len = 80.
Send_to_KernelPeer.............
Send_to_KernelPeer......
Msg from kernel netlink: countRx = 1, payload len = 120, Msg info.hdr.nlmsg_len = 136.
Send_to_KernelPeer.............
Send_to_KernelPeer......
Msg from kernel netlink: countRx = 2, payload len = 120, Msg info.hdr.nlmsg_len = 136.
Send_to_KernelPeer.............
Send_to_KernelPeer......
Msg from kernel netlink: countRx = 3, payload len = 120, Msg info.hdr.nlmsg_len = 136.
Send_to_KernelPeer.............
Send_to_KernelPeer......
Msg from kernel netlink: countRx = 4, payload len = 120, Msg info.hdr.nlmsg_len = 136.
Send_to_KernelPeer.............
Send_to_KernelPeer......
Msg from kernel netlink: countRx = 5, payload len = 120, Msg info.hdr.nlmsg_len = 136.
Send_to_KernelPeer.............
Send_to_KernelPeer......
Msg from kernel netlink: countRx = 6, payload len = 120, Msg info.hdr.nlmsg_len = 136.
Msg from kernel netlink: countRx = 7, payload len = 64, Msg info.hdr.nlmsg_len = 80.
```

root@jmakg9: /home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Offloading-Observing-Window1

```
Write Rx Ring1 - succeed 120.
Read Tx Ring1: Size = 120.
Write Rx Ring1 - start 120.
Write Rx Ring1 - succeed 120.
Read Tx Ring1: Size = 120.
Write Rx Ring1 - start 120.
Write Rx Ring1 - succeed 120.
Read Tx Ring1: Size = 120.
Write Rx Ring1 - start 120.
Write Rx Ring1 - succeed 120.
Read Tx Ring1: Size = 120.
Write Rx Ring1 - start 120.
Write Rx Ring1 - succeed 120.
Read Tx Ring1: Size = 120.
Write Rx Ring1 - start 120.
Write Rx Ring1 - succeed 120.
Read Tx Ring1: Size = 120.
Write Rx Ring1 - start 120.
Write Rx Ring1 - succeed 120.
Read Tx Ring1: Size = 120.
Write Rx Ring1 - start 120.
Write Rx Ring1 - succeed 120.
Read Tx Ring1: Size = 120.
```

root@jmakg9: /home/jmakg9/Desktop/EPC9pclock/VirtualNetDevice/Offloading-Observing-Window2

```
Write Rx Ring2 - succeed 120.
Read Tx Ring2: Size = 120.
Write Rx Ring2 - start 120.
Write Rx Ring2 - succeed 120.
Read Tx Ring2: Size = 120.
Write Rx Ring2 - start 120.
Write Rx Ring2 - succeed 120.
Read Tx Ring2: Size = 120.
Write Rx Ring2 - start 120.
Write Rx Ring2 - succeed 120.
Read Tx Ring2: Size = 120.
Write Rx Ring2 - start 120.
Write Rx Ring2 - succeed 120.
Read Tx Ring2: Size = 120.
Write Rx Ring2 - start 120.
Write Rx Ring2 - succeed 120.
Read Tx Ring2: Size = 120.
Write Rx Ring2 - start 120.
Write Rx Ring2 - succeed 120.
Read Tx Ring2: Size = 120.
Write Rx Ring2 - start 120.
Write Rx Ring2 - succeed 120.
Read Tx Ring2: Size = 120.
```

FIGURE 30. The shooting screens related to the startup information of Virtual-Upper-MAC, Offloading-Engine, Offloading-Observing-Window1, and Offloading-Observing-Window2 in software defined open HetNets platform.

Rx Ring2" in yellow rectangle -> "Send_to_KernelPeer" in light green rectangle -> Virtual Upper MAC -> Upper Protocol Stack) and sent the reply packets to UE (i.e., Upper Protocol Stack -> Virtual Upper MAC -> "Msg from kernel netlink" in light green rectangle -> "Read Tx Ring1" in pink rectangle and "Read Tx Ring2" in yellow rectangle).

Till now, we have done the packets transmission experiment and we can draw the conclusion the current version open HetNets platform can work correctly. Note that there are two virtual PHY thread (i.e., Offloading Observing Window1 and Offloading Observing Window2) in our current version HetNets platform, and each of them is one-to-one match to an available signal defined in Linux system. Besides, as a matter of experience, we can say that the more threads run, the better throughput achieved by HetNets platform. However, when the thread number larger than a specific value, the system performance will be degraded with the reason that there will be causing tremendous overhead by running numerous threads. Therefore, it is necessary to figure out the optimal thread number through performance testing experiment.

As shown in Fig.15, there are 31 signals can be used by user thread (i.e., from 34-64). Based on this principle, we gradually increase the virtual PHY thread number from 2 – 30 by step length 2, then we respectively evaluate each version HetNets platform. For instance, under the condition virtual PHY thread number equals to 4, we use the throughput testing tool "iperf" to conduct the performance testing experiment and collect the experiment data obtained by those two ethernet

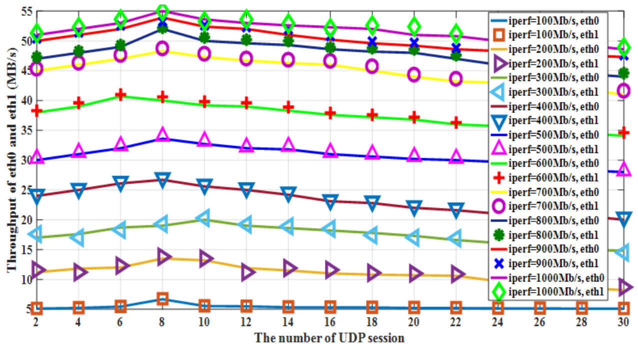


FIGURE 31. Practical downlink throughputs achieved by eth0 (lines) and eth1(markers) under different number of UDP sessions (i.e., the virtual PHY thread).

interfaces (i.e., eth0 and eth1 shown in Fig.24). We conduct ten times performance testing experiment and then we can draw the following statistic figure shown in Fig.31.

As seen in Fig.31, the practical downlink throughput volume increase with increasing the theoretical downlink throughput (i.e., from "iperf = 100 Mbit/s" to "iperf = 1000 Mbit/s"). For example, when we set the theoretical throughput to 600Mbits/s (i.e., "iperf = 600Mb/s"), the practical throughputs obtained by eth0 and eth1 are respectively 37Mbytes/s, which is almost equal to the theoretical downlink throughput. From Fig.31, we can observe that each curve follows a trend that when the virtual PHY thread number lower

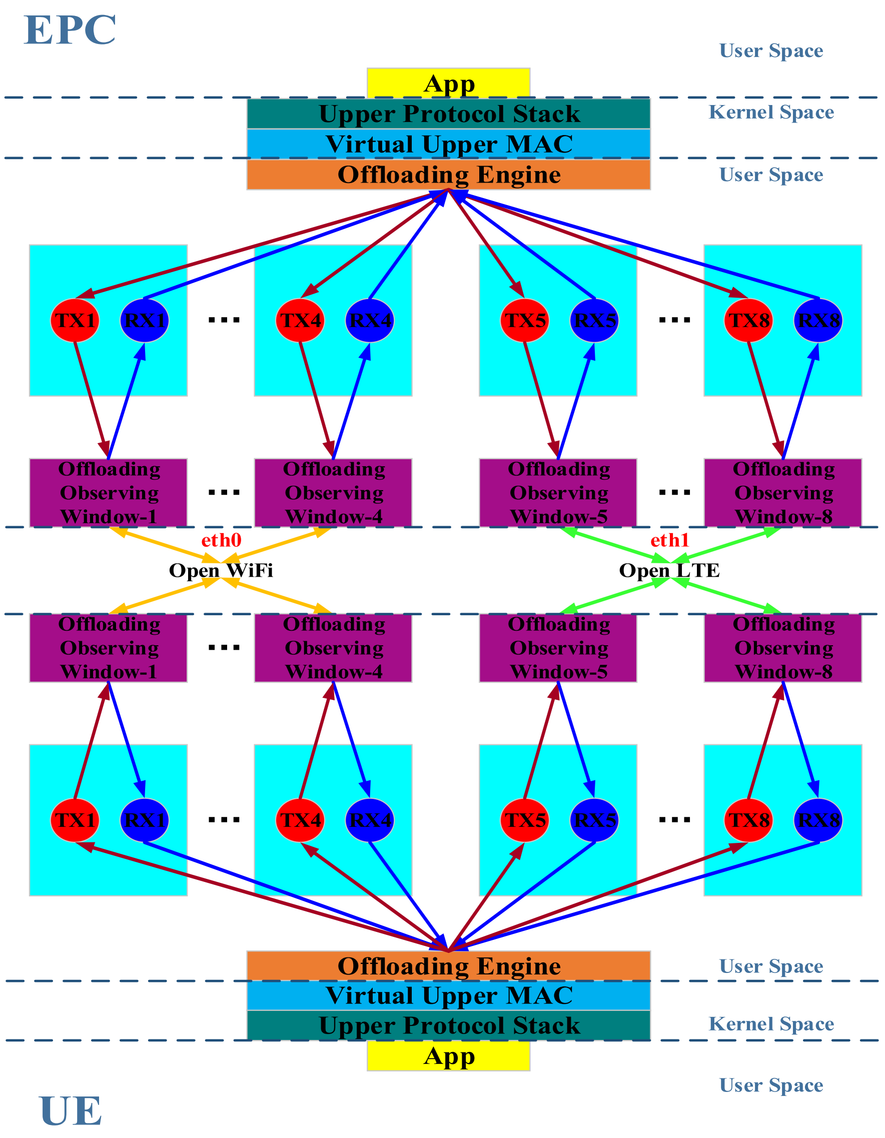
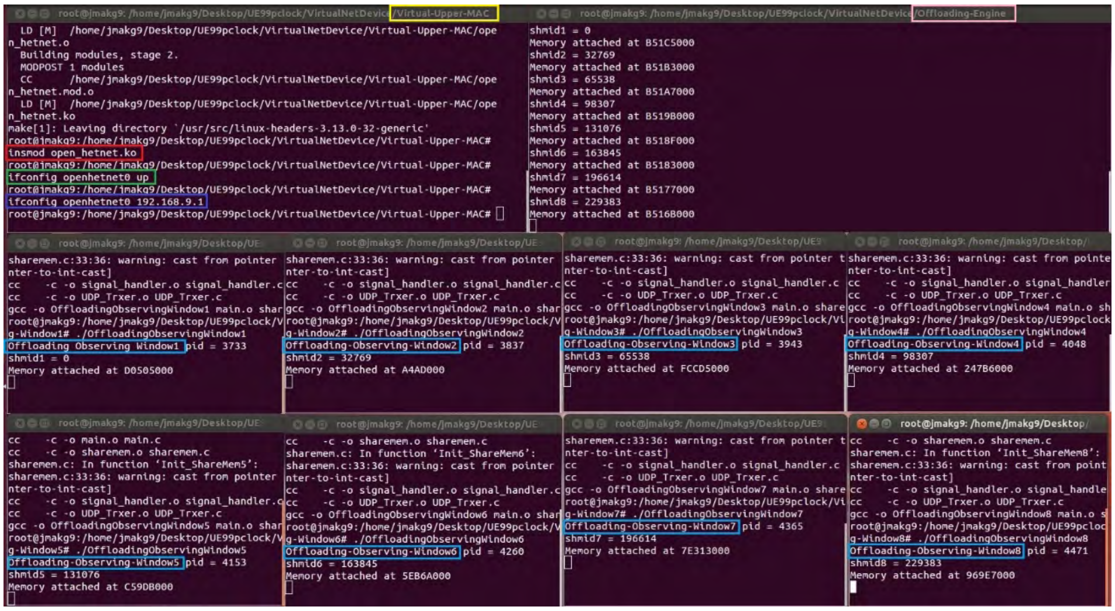


FIGURE 32. The optimized software defined open HetNets platform.

than 8, the throughput increases with increasing the virtual PHY thread; however, when the virtual PHY thread number larger than 8, the throughput decreases with continuing to increase the virtual PHY thread. Therefore, we can conclude that the HetNets platform has the best performance when the virtual PHY thread number equals to 8.

Based on the above-mentioned analyses, we re-design the HetNets platform's software architecture. As shown in Fig.32, we implement 8 virtual PHY threads, the first four virtual PHY threads (i.e., from 1 - 4) are responsible for communicating with open WiFi platform through eth0 and the last four virtual PHY threads (i.e., from 5 - 8) are responsible for communicating with open LTE platform through eth1. Moreover, we can note that there are eight groups TX rings and RX rings, the first four sets are correspondent to the first four virtual PHY threads and the last four sets are correspondent to the last four virtual PHY threads. For each sending packet transmitted from the virtual upper MAC, the Offloading Engine module uses equation ‘‘counter%8’’ to decide which path to deliver this packet. For example, if the remainder equals to (1-7), then the packet will be put into TX (1-7), if the remainder equals to 0, then packet will be put into TX8. In the receiver, we do the inverse process for the arrival packets. Furthermore, in order to guarantee the in-sequence reception, we allocate a large array in Offloading Engine module. Then, those received packets will be first buffered in this array and then be sorted before they submitted to the virtual upper MAC.

Based on the optimized architecture shown in Fig.32, we carry out the packet transmission experiment to validate the correctness of the optimization version HetNets platform. specifically, we conduct the same experiment procedures



```
root@jmakg9: /home/jmakg9/Desktop/UE99pclock/VirtualNetDevice/Virtual-Upper-MAC
  LD [M]  /home/jmakg9/Desktop/UE99pclock/VirtualNetDevice/Virtual-Upper-MAC/ope
n_hetnet.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/jmakg9/Desktop/UE99pclock/VirtualNetDevice/Virtual-Upper-MAC/ope
n_hetnet.mod.o
  LD [M]  /home/jmakg9/Desktop/UE99pclock/VirtualNetDevice/Virtual-Upper-MAC/ope
n_hetnet.ko
make[1]: Leaving directory `/usr/src/linux-headers-3.13.0-32-generic'
root@jmakg9:/home/jmakg9/Desktop/UE99pclock/VirtualNetDevice/Virtual-Upper-MAC#
insmod open_hetnet.ko
root@jmakg9:/home/jmakg9/Desktop/UE99pclock/VirtualNetDevice/Virtual-Upper-MAC#
ifconfig openhetnet0 up
root@jmakg9:/home/jmakg9/Desktop/UE99pclock/VirtualNetDevice/Virtual-Upper-MAC#
ifconfig openhetnet0 192.168.9.1
root@jmakg9:/home/jmakg9/Desktop/UE99pclock/VirtualNetDevice/Virtual-Upper-MAC#
```

```
root@jmakg9: /home/jmakg9/Desktop/UE99pclock/VirtualNetDevice/Offloading-Engine
shmid1 = 0
Memory attached at B51C5000
shmid2 = 32769
Memory attached at B51B3000
shmid3 = 65538
Memory attached at B51A7000
shmid4 = 98307
Memory attached at B519B000
shmid5 = 131076
Memory attached at B518F000
shmid6 = 163845
Memory attached at B5183000
shmid7 = 196614
Memory attached at B5177000
shmid8 = 229383
Memory attached at B516B000
```

```
root@jmakg9: /home/jmakg9/Desktop/UE
sharemem.c:33:36: warning: cast from pointer
nter-to-int-cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow1 main.o shar
root@jmakg9:/home/jmakg9/Desktop/UE99pclock/V
g-Window1# ./OffloadingObservingWindow1
Offloading Observing Window1 pid = 3733
shmid1 = 0
Memory attached at D0505000
```

```
root@jmakg9: /home/jmakg9/Desktop/UE
sharemem.c:33:36: warning: cast from pointer
nter-to-int-cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow2 main.o shar
root@jmakg9:/home/jmakg9/Desktop/UE99pclock/V
g-Window2# ./OffloadingObservingWindow2
Offloading-Observing-Window2 pid = 3837
shmid2 = 32769
Memory attached at A4AD000
```

```
root@jmakg9: /home/jmakg9/Desktop/UE
sharemem.c:33:36: warning: cast from pointer t
nter-to-int-cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow3 main.o share
root@jmakg9:/home/jmakg9/Desktop/UE99pclock/Vi
g-Window3# ./OffloadingObservingWindow3
Offloading-Observing-Window3 pid = 3943
shmid3 = 65538
Memory attached at FCCD5000
```

```
root@jmakg9: /home/jmakg9/Desktop/
sharemem.c:33:36: warning: cast from pointe
nter-to-int-cast]
cc    -c -o signal_handler.o signal_handler
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow4 main.o sh
root@jmakg9:/home/jmakg9/Desktop/UE99pclock
g-Window4# ./OffloadingObservingWindow4
Offloading-Observing-Window4 pid = 4048
shmid4 = 98307
Memory attached at 247B6000
```

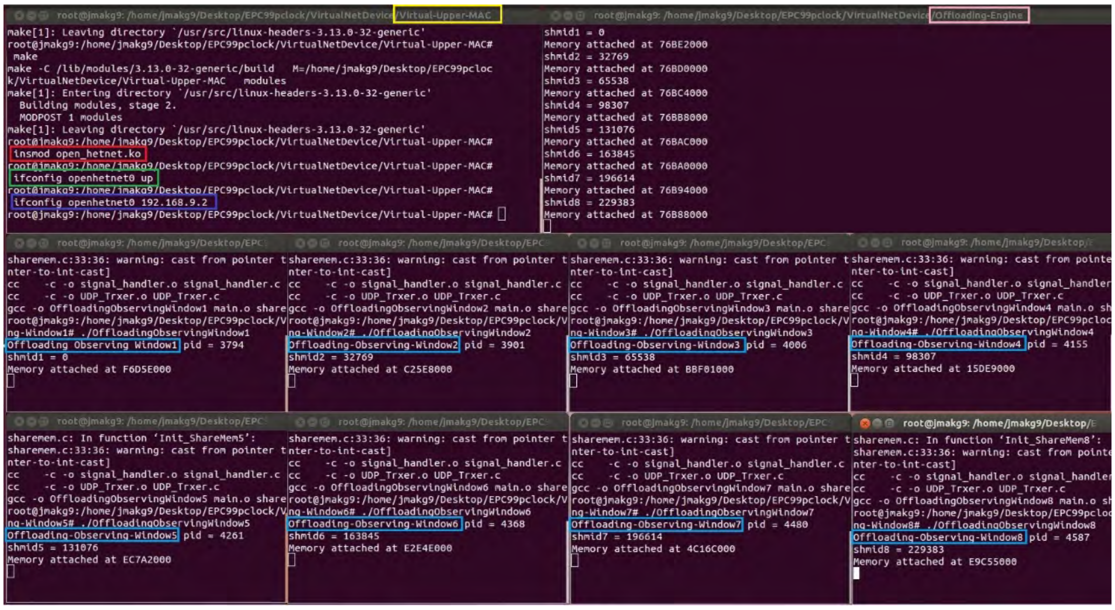
```
root@jmakg9: /home/jmakg9/Desktop/UE
cc    -c -o main.o main.c
cc    -c -o sharemem.o sharemem.c
sharemem.c: In function ‘Init_ShareMem5’:
sharemem.c:33:36: warning: cast from pointer
nter-to-int-cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow5 main.o shar
root@jmakg9:/home/jmakg9/Desktop/UE99pclock/V
g-Window5# ./OffloadingObservingWindow5
Offloading-Observing-Window5 pid = 4153
shmid5 = 131076
Memory attached at C59DB000
```

```
root@jmakg9: /home/jmakg9/Desktop/UE
cc    -c -o sharemem.o sharemem.c
sharemem.c: In function ‘Init_ShareMem6’:
sharemem.c:33:36: warning: cast from pointer
nter-to-int-cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow6 main.o shar
root@jmakg9:/home/jmakg9/Desktop/UE99pclock/V
g-Window6# ./OffloadingObservingWindow6
Offloading-Observing-Window6 pid = 4260
shmid6 = 163845
Memory attached at 5EB6A000
```

```
root@jmakg9: /home/jmakg9/Desktop/UE9
sharemem.c:33:36: warning: cast from pointer t
nter-to-int-cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow7 main.o share
root@jmakg9:/home/jmakg9/Desktop/UE99pclock/Vi
g-Window7# ./OffloadingObservingWindow7
Offloading-Observing-Window7 pid = 4365
shmid7 = 196614
Memory attached at 7E313000
```

```
root@jmakg9: /home/jmakg9/Desktop/
cc    -c -o sharemem.o sharemem.c
sharemem.c: In function ‘Init_ShareMem8’:
sharemem.c:33:36: warning: cast from point
nter-to-int-cast]
cc    -c -o signal_handler.o signal_handle
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow8 main.o s
root@jmakg9:/home/jmakg9/Desktop/UE99pcloc
g-Window8# ./OffloadingObservingWindow8
Offloading-Observing-Window8 pid = 4471
shmid8 = 229383
Memory attached at 969E7000
```

FIGURE 33. The shooting screens related to the initialization information of Virtual-Upper-MAC, Offloading-Engine, and eight Offloading-Observing-Windows (i.e., from1-8) in software defined UE after optimized.



```
root@jmakg9: /home/jmakg9/Desktop/EPC99pclock/VirtualNetDevice/Virtual-Upper-MAC
make[1]: Leaving directory `/usr/src/linux-headers-3.13.0-32-generic'
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/VirtualNetDevice/Virtual-Upper-MAC#
 make
make -C /lib/modules/3.13.0-32-generic/build   M=/home/jmakg9/Desktop/EPC99pcloc
k/VirtualNetDevice/Virtual-Upper-MAC   modules
make[1]: Entering directory `/usr/src/linux-headers-3.13.0-32-generic'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory `/usr/src/linux-headers-3.13.0-32-generic'
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/VirtualNetDevice/Virtual-Upper-MAC#
 insmod open_hetnet.ko
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/VirtualNetDevice/Virtual-Upper-MAC#
 ifconfig openhetnet0 up
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/VirtualNetDevice/Virtual-Upper-MAC#
 ifconfig openhetnet0 192.168.9.2
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/VirtualNetDevice/Virtual-Upper-MAC#
```

```
root@jmakg9: /home/jmakg9/Desktop/EPC99pclock/VirtualNetDevice/Offloading-Engine
shmid1 = 0
Memory attached at 76BE2000
shmid2 = 32769
Memory attached at 76BD0000
shmid3 = 65538
Memory attached at 76BC4000
shmid4 = 98307
Memory attached at 76BB8000
shmid5 = 131076
Memory attached at 76BAC000
shmid6 = 163845
Memory attached at 76BA0000
shmid7 = 196614
Memory attached at 76B94000
shmid8 = 229383
Memory attached at 76B88000
```

```
root@jmakg9: /home/jmakg9/Desktop/EPC
sharemem.c:33:36: warning: cast from pointer t
nter-to-int-cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow1 main.o share
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/V
ng-Window1# ./OffloadingObservingWindow1
Offloading Observing Window1 pid = 3794
shmid1 = 0
Memory attached at F6D5E000
```

```
root@jmakg9: /home/jmakg9/Desktop/EPC
sharemem.c:33:36: warning: cast from pointer t
nter-to-int-cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow2 main.o share
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/V
ng-Window2# ./OffloadingObservingWindow2
Offloading-Observing-Window2 pid = 3901
shmid2 = 32769
Memory attached at C25E8000
```

```
root@jmakg9: /home/jmakg9/Desktop/EPC
sharemem.c:33:36: warning: cast from pointer t
nter-to-int-cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow3 main.o share
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/V
ng-Window3# ./OffloadingObservingWindow3
Offloading-Observing-Window3 pid = 4006
shmid3 = 65538
Memory attached at BBF01000
```

```
root@jmakg9: /home/jmakg9/Desktop/
sharemem.c:33:36: warning: cast from pointe
nter-to-int-cast]
cc    -c -o signal_handler.o signal_handler
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow4 main.o sh
root@jmakg9:/home/jmakg9/Desktop/EPC99pcloc
ng-Window4# ./OffloadingObservingWindow4
Offloading-Observing-Window4 pid = 4155
shmid4 = 98307
Memory attached at 15DE9000
```

```
root@jmakg9: /home/jmakg9/Desktop/EPC
sharemem.c: In function ‘Init_ShareMem5’:
sharemem.c:33:36: warning: cast from pointer t
nter-to-int-cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow5 main.o share
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/V
ng-Window5# ./OffloadingObservingWindow5
Offloading-Observing-Window5 pid = 4261
shmid5 = 131076
Memory attached at EC7A2000
```

```
root@jmakg9: /home/jmakg9/Desktop/EPC
sharemem.c:33:36: warning: cast from pointer t
nter-to-int-cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow6 main.o share
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/V
ng-Window6# ./OffloadingObservingWindow6
Offloading-Observing-Window6 pid = 4368
shmid6 = 163845
Memory attached at E2E4E000
```

```
root@jmakg9: /home/jmakg9/Desktop/EPC
sharemem.c:33:36: warning: cast from pointer t
nter-to-int-cast]
cc    -c -o signal_handler.o signal_handler.c
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow7 main.o share
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/V
ng-Window7# ./OffloadingObservingWindow7
Offloading-Observing-Window7 pid = 4480
shmid7 = 196614
Memory attached at 4C16C000
```

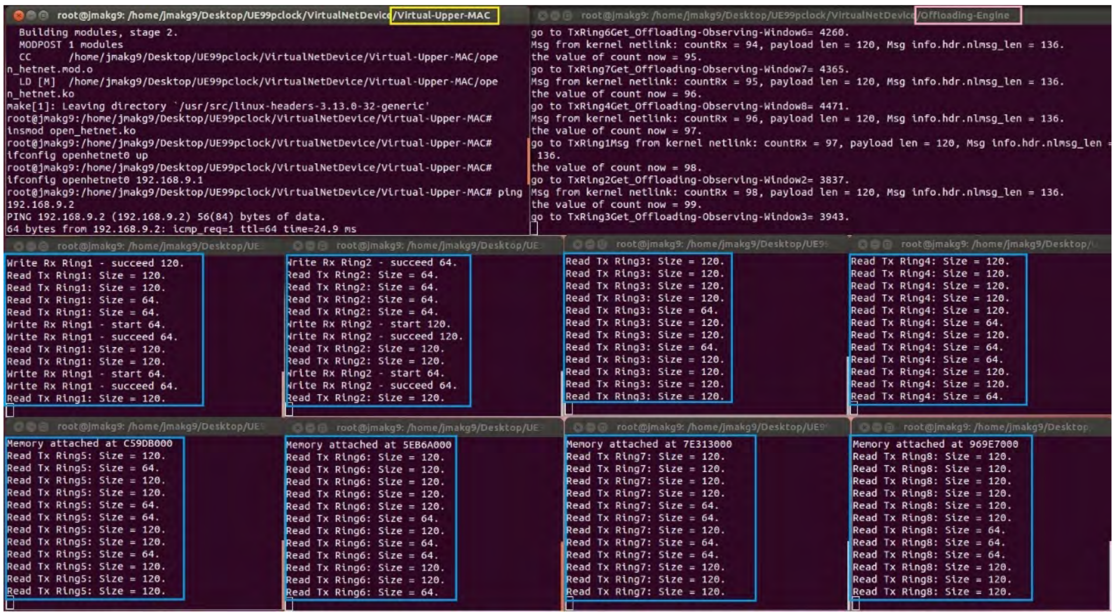
```
root@jmakg9: /home/jmakg9/Desktop/E
sharemem.c: In function ‘Init_ShareMem8’:
sharemem.c:33:36: warning: cast from pointe
nter-to-int-cast]
cc    -c -o signal_handler.o signal_handler
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow8 main.o sh
root@jmakg9:/home/jmakg9/Desktop/EPC99pcloc
ng-Window8# ./OffloadingObservingWindow8
Offloading-Observing-Window8 pid = 4587
shmid8 = 229383
Memory attached at E9C55000
```

FIGURE 34. The shooting screens related to the initialization information of Virtual-Upper-MAC, Offloading-Engine, and eight Offloading-Observing-Windows in software defined open HetNets platform after optimized.

as described previously. Then we respectively collect some milestone shooting screens from software-defined HetNets platform and software-defined UE.

Fig.33 and Fig.34 respectively demonstrate the shooting screens in soft UE and in HetNets platform, the yellow rectangle highlights the Virtual Upper MAC window, the pink rectangle highlights the Offloading Engine window, and those eight light blue rectangles highlight eight Offloading Observing Windows. Note that the IP address of soft UE is set to "192.168.9.1" and the IP address of HetNets platform is configured to "192.168.9.2". According to these screens information, we can learn that the soft UE and the HetNets platform have been successfully initialized. Then, in UE side, we input command "ping 192.168.9.2" to send packets to HetNets platform. After then, we can obtain another two figures shown in Fig.35 and Fig.36.

As depicted in Fig.35, eight light blue rectangles represent eight concurrent transmission/reception paths in UE. For example, the message "Read Tx Ring" in the light blue rectangle shows the transmission path of the ARP request packet and the ICMP request packets, and the message "Write Rx Ring" demonstrates the reception of the ARP



```
root@jmakg9: /home/jmakg9/Desktop/UE99pclock/VirtualNetDevice/Virtual-Upper-MAC
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/jmakg9/Desktop/UE99pclock/VirtualNetDevice/Virtual-Upper-MAC/ope
n_hetnet.mod.o
  LD [M]  /home/jmakg9/Desktop/UE99pclock/VirtualNetDevice/Virtual-Upper-MAC/ope
n_hetnet.ko
make[1]: Leaving directory `/usr/src/linux-headers-3.13.0-32-generic'
root@jmakg9:/home/jmakg9/Desktop/UE99pclock/VirtualNetDevice/Virtual-Upper-MAC#
insmod open_hetnet.ko
root@jmakg9:/home/jmakg9/Desktop/UE99pclock/VirtualNetDevice/Virtual-Upper-MAC#
ifconfig openhetnet0 up
root@jmakg9:/home/jmakg9/Desktop/UE99pclock/VirtualNetDevice/Virtual-Upper-MAC#
ifconfig openhetnet0 192.168.9.1
root@jmakg9:/home/jmakg9/Desktop/UE99pclock/VirtualNetDevice/Virtual-Upper-MAC# ping
192.168.9.2
PING 192.168.9.2 (192.168.9.2) 56(84) bytes of data.
64 bytes from 192.168.9.2: icmp_req=1 ttl=64 time=24.9 ms
```

```
root@jmakg9: /home/jmakg9/Desktop/UE99pclock/VirtualNetDevice/Offloading-Engine
go to TxRing6Get_Offloading-Observing-Window6= 4260.
Msg from kernel netlink: countRx = 94, payload len = 120, Msg info.hdr.nlmsg_len = 136.
the value of count now = 95.
go to TxRing7Get_Offloading-Observing-Window7= 4365.
Msg from kernel netlink: countRx = 95, payload len = 120, Msg info.hdr.nlmsg_len = 136.
the value of count now = 96.
go to TxRing4Get_Offloading-Observing-Window8= 4471.
Msg from kernel netlink: countRx = 96, payload len = 120, Msg info.hdr.nlmsg_len = 136.
the value of count now = 97.
go to TxRing1Msg from kernel netlink: countRx = 97, payload len = 120, Msg info.hdr.nlmsg_len =
 136.
the value of count now = 98.
go to TxRing2Get_Offloading-Observing-Window2= 3837.
Msg from kernel netlink: countRx = 98, payload len = 120, Msg info.hdr.nlmsg_len = 136.
the value of count now = 99.
go to TxRing3Get_Offloading-Observing-Window3= 3943.
```

```
root@jmakg9: /home/jmakg9/Desktop/UE
Write Rx Ring1 - succeed 120.
Read Tx Ring1: Size = 120.
Read Tx Ring1: Size = 120.
Read Tx Ring1: Size = 64.
Read Tx Ring1: Size = 64.
Write Rx Ring1 - start 64.
Write Rx Ring1 - succeed 64.
Read Tx Ring1: Size = 120.
Read Tx Ring1: Size = 120.
Write Rx Ring1 - start 64.
Write Rx Ring1 - succeed 64.
Read Tx Ring1: Size = 120.
```

```
root@jmakg9: /home/jmakg9/Desktop/UE
Write Rx Ring2 - succeed 64.
Read Tx Ring2: Size = 64.
Read Tx Ring2: Size = 64.
Read Tx Ring2: Size = 64.
Read Tx Ring2: Size = 64.
Write Rx Ring2 - start 120.
Write Rx Ring2 - succeed 120.
Read Tx Ring2: Size = 120.
Read Tx Ring2: Size = 120.
Write Rx Ring2 - start 64.
Write Rx Ring2 - succeed 64.
Read Tx Ring2: Size = 120.
```

```
root@jmakg9: /home/jmakg9/Desktop/UE9
Read Tx Ring3: Size = 120.
Read Tx Ring3: Size = 120.
Read Tx Ring3: Size = 120.
Read Tx Ring3: Size = 120.
Read Tx Ring3: Size = 64.
Read Tx Ring3: Size = 120.
Read Tx Ring3: Size = 120.
Read Tx Ring3: Size = 64.
Read Tx Ring3: Size = 120.
Read Tx Ring3: Size = 120.
Read Tx Ring3: Size = 120.
Read Tx Ring3: Size = 120.
```

```
root@jmakg9: /home/jmakg9/Desktop/U
Read Tx Ring4: Size = 120.
Read Tx Ring4: Size = 120.
Read Tx Ring4: Size = 120.
Read Tx Ring4: Size = 120.
Read Tx Ring4: Size = 120.
Read Tx Ring4: Size = 64.
Read Tx Ring4: Size = 120.
Read Tx Ring4: Size = 64.
Read Tx Ring4: Size = 64.
Read Tx Ring4: Size = 120.
Read Tx Ring4: Size = 120.
Read Tx Ring4: Size = 64.
```

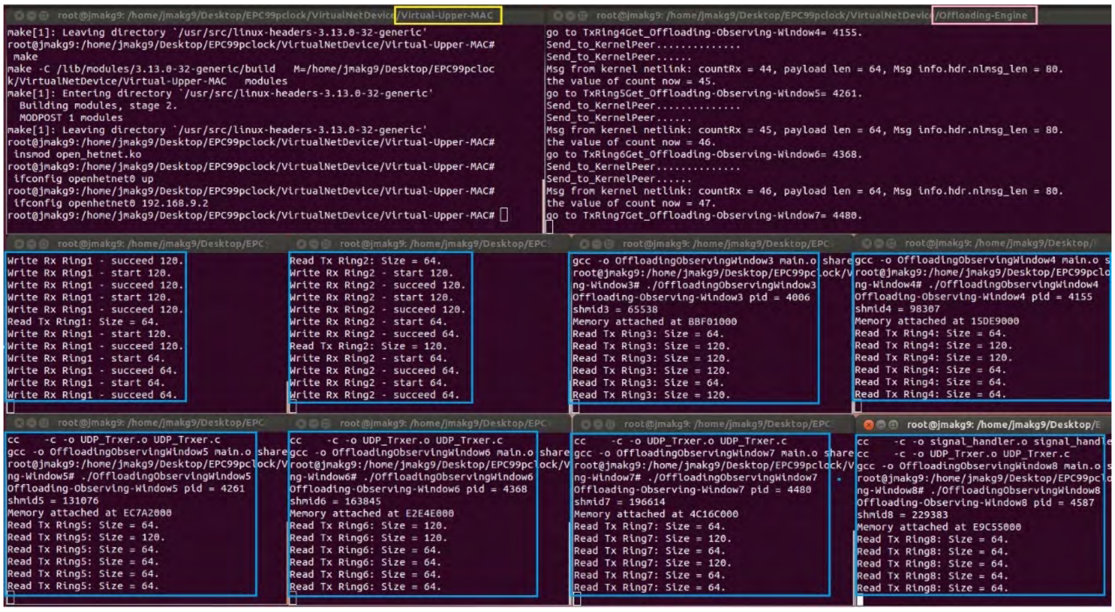
```
root@jmakg9: /home/jmakg9/Desktop/UE
Memory attached at C59DB000
Read Tx Ring5: Size = 120.
Read Tx Ring5: Size = 64.
Read Tx Ring5: Size = 120.
Read Tx Ring5: Size = 120.
Read Tx Ring5: Size = 64.
Read Tx Ring5: Size = 64.
Read Tx Ring5: Size = 120.
Read Tx Ring5: Size = 120.
Read Tx Ring5: Size = 64.
Read Tx Ring5: Size = 120.
Read Tx Ring5: Size = 120.
Read Tx Ring5: Size = 120.
```

```
root@jmakg9: /home/jmakg9/Desktop/UE
Memory attached at 5EB6A000
Read Tx Ring6: Size = 120.
Read Tx Ring6: Size = 120.
Read Tx Ring6: Size = 120.
Read Tx Ring6: Size = 120.
Read Tx Ring6: Size = 120.
Read Tx Ring6: Size = 64.
Read Tx Ring6: Size = 120.
Read Tx Ring6: Size = 64.
Read Tx Ring6: Size = 64.
Read Tx Ring6: Size = 120.
Read Tx Ring6: Size = 120.
Read Tx Ring6: Size = 64.
```

```
root@jmakg9: /home/jmakg9/Desktop/UE9
Memory attached at 7E313000
Read Tx Ring7: Size = 120.
Read Tx Ring7: Size = 120.
Read Tx Ring7: Size = 120.
Read Tx Ring7: Size = 120.
Read Tx Ring7: Size = 64.
Read Tx Ring7: Size = 64.
Read Tx Ring7: Size = 120.
Read Tx Ring7: Size = 64.
Read Tx Ring7: Size = 64.
Read Tx Ring7: Size = 120.
Read Tx Ring7: Size = 120.
Read Tx Ring7: Size = 120.
```

```
root@jmakg9: /home/jmakg9/Desktop
Memory attached at 969E7000
Read Tx Ring8: Size = 120.
Read Tx Ring8: Size = 120.
Read Tx Ring8: Size = 120.
Read Tx Ring8: Size = 120.
Read Tx Ring8: Size = 120.
Read Tx Ring8: Size = 120.
Read Tx Ring8: Size = 64.
Read Tx Ring8: Size = 64.
Read Tx Ring8: Size = 64.
Read Tx Ring8: Size = 120.
Read Tx Ring8: Size = 120.
Read Tx Ring8: Size = 120.
```

FIGURE 35. The shooting screens related to the startup information of Virtual-Upper-MAC, Offloading-Engine, and eight Offloading-Observing-Windows in software defined UE after optimized.



```
root@jmakg9: /home/jmakg9/Desktop/EPC99pclock/VirtualNetDevice/Virtual-Upper-MAC
make[1]: Leaving directory `/usr/src/linux-headers-3.13.0-32-generic'
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/VirtualNetDevice/Virtual-Upper-MAC#
 make
make -C /lib/modules/3.13.0-32-generic/build  M=/home/jmakg9/Desktop/EPC99pcloc
k/VirtualNetDevice/Virtual-Upper-MAC  modules
make[1]: Entering directory `/usr/src/linux-headers-3.13.0-32-generic'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory `/usr/src/linux-headers-3.13.0-32-generic'
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/VirtualNetDevice/Virtual-Upper-MAC#
 insmod open_hetnet.ko
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/VirtualNetDevice/Virtual-Upper-MAC#
 ifconfig openhetnet0 up
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/VirtualNetDevice/Virtual-Upper-MAC#
 ifconfig openhetnet0 192.168.9.2
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/VirtualNetDevice/Virtual-Upper-MAC#
```

```
root@jmakg9: /home/jmakg9/Desktop/EPC99pclock/VirtualNetDevice/Offloading-Engine
go to TxRing4Get_Offloading-Observing-Window4= 4155.
Send_to_KernelPeer.............
Send_to_KernelPeer......
Msg from kernel netlink: countRx = 44, payload len = 64, Msg info.hdr.nlmsg_len = 80.
the value of count now = 45.
go to TxRing5Get_Offloading-Observing-Window5= 4261.
Send_to_KernelPeer.............
Send_to_KernelPeer......
Msg from kernel netlink: countRx = 45, payload len = 64, Msg info.hdr.nlmsg_len = 80.
the value of count now = 46.
go to TxRing6Get_Offloading-Observing-Window6= 4368.
Send_to_KernelPeer.............
Send_to_KernelPeer......
Msg from kernel netlink: countRx = 46, payload len = 64, Msg info.hdr.nlmsg_len = 80.
the value of count now = 47.
go to TxRing7Get_Offloading-Observing-Window7= 4480.
```

```
root@jmakg9: /home/jmakg9/Desktop/EPC
Write Rx Ring1 - succeed 120.
Write Rx Ring1 - start 120.
Write Rx Ring1 - succeed 120.
Write Rx Ring1 - start 120.
Write Rx Ring1 - succeed 120.
Read Tx Ring1: Size = 64.
Write Rx Ring1 - start 120.
Write Rx Ring1 - succeed 120.
Write Rx Ring1 - start 64.
Write Rx Ring1 - succeed 64.
Write Rx Ring1 - start 64.
Write Rx Ring1 - succeed 64.
```

```
root@jmakg9: /home/jmakg9/Desktop/EPC
Read Tx Ring2: Size = 64.
Write Rx Ring2 - start 120.
Write Rx Ring2 - succeed 120.
Write Rx Ring2 - start 120.
Write Rx Ring2 - succeed 120.
Write Rx Ring2 - start 64.
Write Rx Ring2 - succeed 64.
Read Tx Ring2: Size = 120.
Write Rx Ring2 - start 64.
Write Rx Ring2 - succeed 64.
Write Rx Ring2 - start 64.
Write Rx Ring2 - succeed 64.
```

```
root@jmakg9: /home/jmakg9/Desktop/EPC
gcc -o OffloadingObservingWindow3 main.o share
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/V
ng-Window3# ./OffloadingObservingWindow3
Offloading-Observing-Window3 pid = 4006
shmid3 = 65538
Memory attached at BBF01000
Read Tx Ring3: Size = 64.
Read Tx Ring3: Size = 120.
Read Tx Ring3: Size = 120.
Read Tx Ring3: Size = 120.
Read Tx Ring3: Size = 64.
Read Tx Ring3: Size = 120.
```

```
root@jmakg9: /home/jmakg9/Desktop/
gcc -o OffloadingObservingWindow4 main.o s
root@jmakg9:/home/jmakg9/Desktop/EPC99pclo
ng-Window4# ./OffloadingObservingWindow4
Offloading-Observing-Window4 pid = 4155
shmid4 = 98307
Memory attached at 15DE9000
Read Tx Ring4: Size = 64.
Read Tx Ring4: Size = 120.
Read Tx Ring4: Size = 120.
Read Tx Ring4: Size = 64.
Read Tx Ring4: Size = 64.
Read Tx Ring4: Size = 64.
```

```
root@jmakg9: /home/jmakg9/Desktop/EPC
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow5 main.o share
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/V
ng-Window5# ./OffloadingObservingWindow5
Offloading-Observing-Window5 pid = 4261
shmid5 = 131076
Memory attached at EC7A2000
Read Tx Ring5: Size = 64.
Read Tx Ring5: Size = 120.
Read Tx Ring5: Size = 64.
Read Tx Ring5: Size = 64.
Read Tx Ring5: Size = 64.
Read Tx Ring5: Size = 64.
```

```
root@jmakg9: /home/jmakg9/Desktop/EPC
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow6 main.o share
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/V
ng-Window6# ./OffloadingObservingWindow6
Offloading-Observing-Window6 pid = 4368
shmid6 = 163845
Memory attached at E2E4E000
Read Tx Ring6: Size = 120.
Read Tx Ring6: Size = 120.
Read Tx Ring6: Size = 64.
Read Tx Ring6: Size = 64.
Read Tx Ring6: Size = 64.
Read Tx Ring6: Size = 64.
```

```
root@jmakg9: /home/jmakg9/Desktop/EPC
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow7 main.o share
root@jmakg9:/home/jmakg9/Desktop/EPC99pclock/V
ng-Window7# ./OffloadingObservingWindow7
Offloading-Observing-Window7 pid = 4480
shmid7 = 196614
Memory attached at 4C16C000
Read Tx Ring7: Size = 64.
Read Tx Ring7: Size = 120.
Read Tx Ring7: Size = 64.
Read Tx Ring7: Size = 120.
Read Tx Ring7: Size = 64.
Read Tx Ring7: Size = 64.
```

```
root@jmakg9: /home/jmakg9/Desktop/E
cc    -c -o signal_handler.o signal_handle
cc    -c -o UDP_Trxer.o UDP_Trxer.c
gcc -o OffloadingObservingWindow8 main.o s
root@jmakg9:/home/jmakg9/Desktop/EPC99pclo
ng-Window8# ./OffloadingObservingWindow8
Offloading-Observing-Window8 pid = 4587
shmid8 = 229383
Memory attached at E9C55000
Read Tx Ring8: Size = 64.
Read Tx Ring8: Size = 64.
Read Tx Ring8: Size = 64.
Read Tx Ring8: Size = 64.
Read Tx Ring8: Size = 64.
```

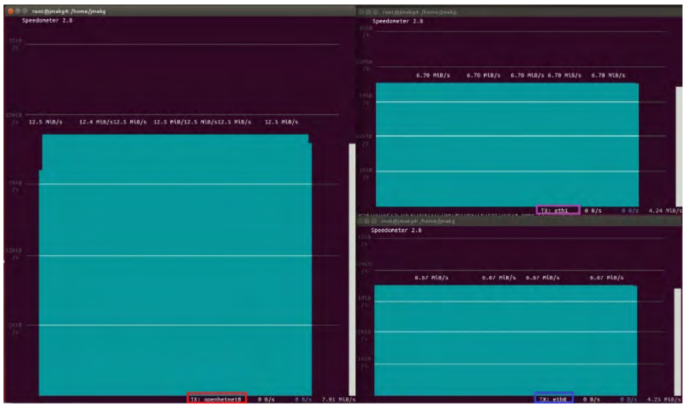
FIGURE 36. The shooting screens related to the startup information of Virtual-Upper-MAC, Offloading-Engine, and eight Offloading-Observing-Windows in software defined open HetNets platform after optimized.

answer packets and the ICMP rely packets. Similarly, eight light blue rectangles in Fig.36 represent eight concurrent transmission/reception paths in HetNets platform, those eight concurrent transmission/reception paths carry out the inverse processes opposite to the soft UE.

F. SOFTWARE DEFINED OPEN HETNETS PLATFORM EVALUATION

In section III.E, we have conducted the optimization process for the open HetNets platform. As illustrated in Fig.24, since the HetNets platform is composed by the software-defined open WiFi platform, the software-defined open LTE platform, and the enhanced EPC which is located in the server. Therefore, it is necessary to carry out the evaluation process for this HetNets platform (i.e., the dash line rectangle shown in Fig.24). In another word, in this section, we want utilize the pressure testing to check out the actual maximum downlink throughput that can be supported by the HetNets platform.

Theoretically speaking, there will be a discrepancy between the theoretical downlink throughput and the actual downlink throughput obtained by the open WiFi platform and the open LTE platform with the reason that there will be



```
root@jmakg4: /home/jmakg
Speedometer 2.8
12.5 MiB/s      12.4 MiB/s12.5 MiB/s  12.5 MiB/12.5 MiB/s12.5 MiB/s      12.5 MiB/s
TX: openhetnet0   0 B/s   0 B/s   7.91 MiB/s

root@jmakg4: /home/jmakg
Speedometer 2.8
6.70 MiB/s      6.70 MiB/s    6.70 MiB/s 6.70 MiB/s    6.70 MiB/s
TX: eth1   0 B/s   0 B/s   4.24 MiB/s

root@jmakg4: /home/jmakg
Speedometer 2.8
6.67 MiB/s        6.67 MiB/s   6.67 MiB/s         6.67 MiB/s
TX: eth0   0 B/s   0 B/s   4.25 MiB/s
```

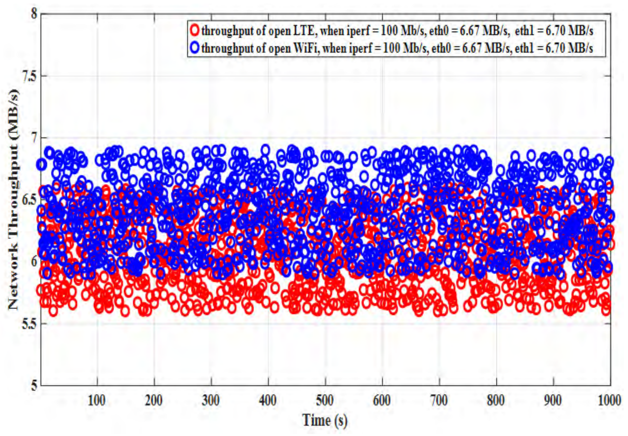
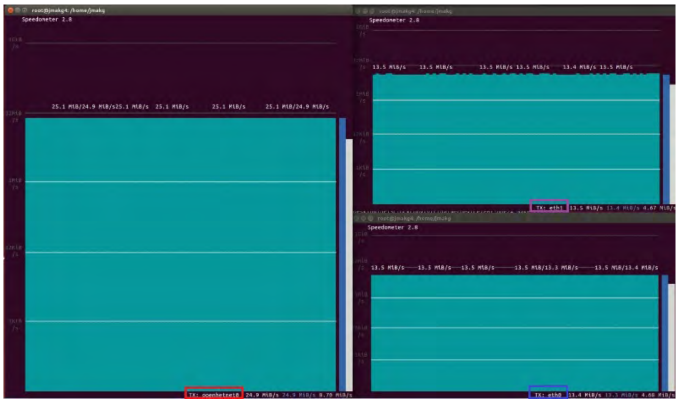


FIGURE 37. The actual throughput obtained by the open WiFi platform and the open LTE platform when the theoretical downlink throughput is set to 100Mbits/s (i.e., "iperf = 100 Mbit/s"), openhetnet0 (i.e., the virtual interface) = 12.50MBytes/s (captured by speedometer), eth0 = 6.67MBytes/s (captured by speedometer), and eth1 =6.70 MBytes/s (captured by speedometer).



```
root@jmakg4: /home/jmakg
Speedometer 2.8
25.1 MiB/24.9 MiB/s25.1 MiB/s  25.1 MiB/s       25.1 MiB/s      25.1 MiB/24.9 MiB/s
TX: openhetnet0  24.9 MiB/s  24.9 MiB/s  8.79 MiB/s

root@jmakg4: /home/jmakg
Speedometer 2.8
13.5 MiB/s     13.5 MiB/s        13.5 MiB/s 13.5 MiB/s     13.4 MiB/s 13.5 MiB/s
TX: eth1  13.5 MiB/s  13.4 MiB/s  4.67 MiB/s

root@jmakg4: /home/jmakg
Speedometer 2.8
13.5 MiB/s—13.5 MiB/s—13.5 MiB/s——13.5 MiB/13.3 MiB/s——13.5 MiB/13.4 MiB/s
TX: eth0  13.4 MiB/s  13.3 MiB/s  4.68 MiB/s
```

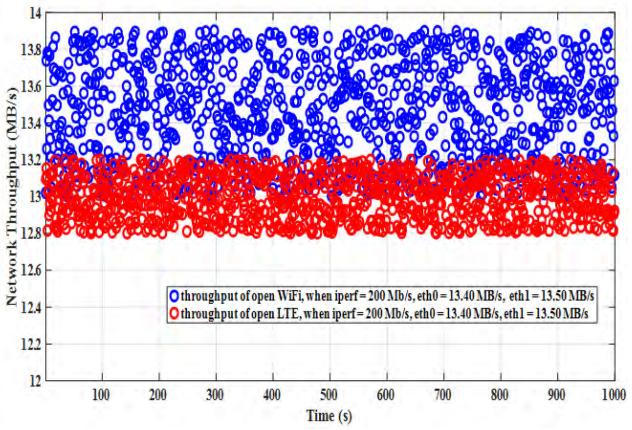


FIGURE 38. The actual throughput obtained by the open WiFi platform and the open LTE platform when the theoretical downlink throughput is set to 200Mbits/s (i.e., "iperf = 200 Mbit/s"), openhetnet0 (i.e., the virtual interface) = 25.10MBytes/s (captured by speedometer), eth0 = 13.40MByte/s (captured by speedometer), and eth1 = 13.50 MBytes/s (captured by speedometer).

some throughput lost incurred by the current version WiFi platform's ability and the current version LTE platform's ability.

To carry out the evaluation experiment, we give the following experiment procedures:

(1). Based on the experiment plan shown in Fig.24, in the server side, we first compile and run the Virtual Upper MAC module, the Offloading Engine module, and eight Offloading Observing Window modules. We then run the speedometer, which is the network throughput observation tool, to observe the practical throughput obtained by openhetnet0, eth0, and eth1.

(2). We respectively compile and run the software-defined open WiFi platform and the software-defined open LTE platform. Then, in these two startup platforms, we respectively input the command "iperf –u –s –i 1" to run iperf, which is the network performance testing tool, to observe the actual throughput obtained by these two platforms.

(3). Based on the above-mentioned procedures, in the server side, we conduct ten sets of experiments by setting ten different theoretical downlink throughputs. To be specific, in the first group experiment, we input the command "iperf –u –c 192.168.9.1 –b 100Mbs/s –t 1000 –i 1" to set the theoretical downlink throughput equals to 100Mbits/s, then based on the speedometer and the accrual throughput obtained by these two platforms, we can obtain the following experiment statistics information shown in Fig.37

(4). For another nine groups experiments, we conduct the same procedures as that in the first group experiment. After then, we can respectively obtain another nine statistic figures which are shown from Fig.38-Fig.46.

The measured downlink throughputs are depicted in Fig.37 – Fig.46. For each figure, it includes two sub-figures of which the one on the left-side is the theoretical downlink throughput observation screen captured by speedometer, the other on the right-side is the actual downlink throughput obtained by the open WiFi platform and by the open LTE platform.

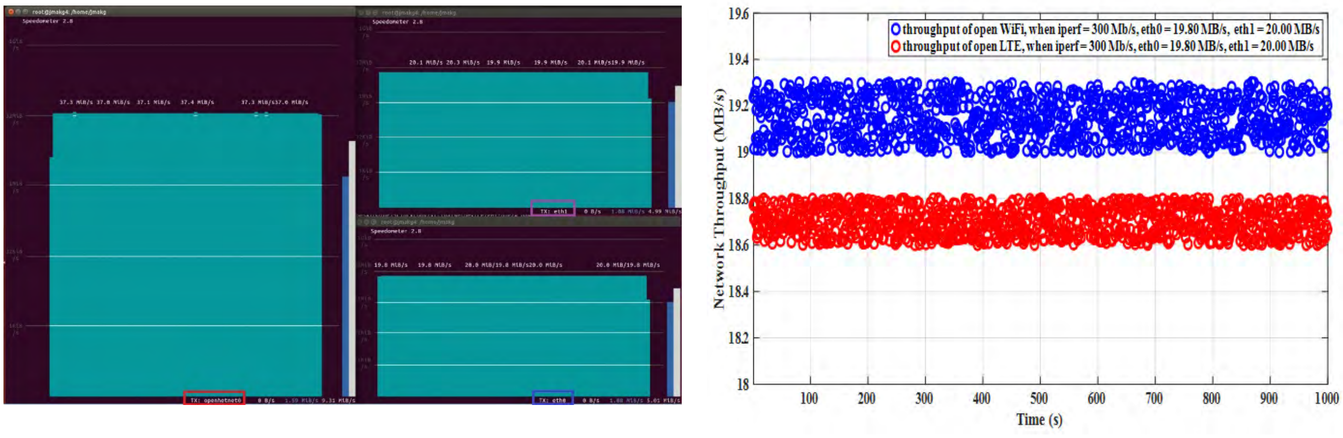


FIGURE 39. The actual throughput obtained by the open WiFi platform and the open LTE platform when the theoretical downlink throughput is set to 300Mbits/s (i.e., "iperf = 300 Mbit/s"), openhetnet0 (i.e., the virtual interface) = 37.20MBytes/s (captured by speedometer), eth0 = 19.80MBytes/s (captured by speedometer), and eth1 = 20.00 MBytes/s (captured by speedometer).

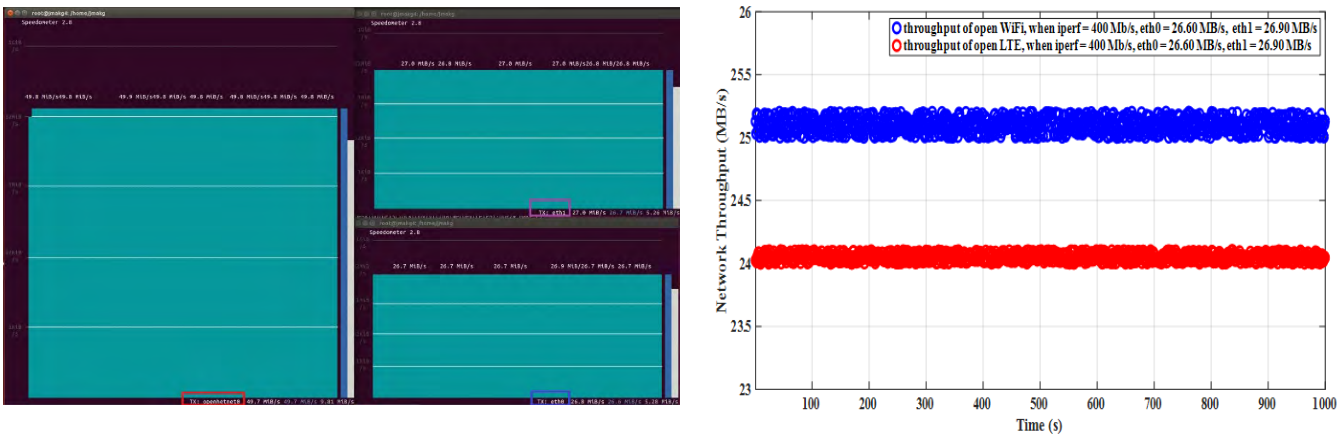
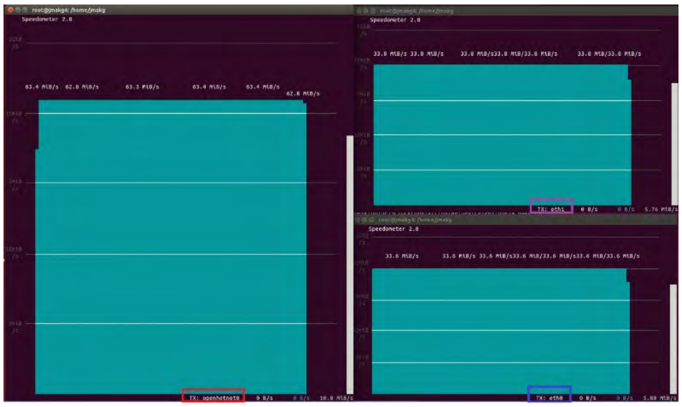


FIGURE 40. The actual throughput obtained by the open WiFi platform and the open LTE platform when the theoretical downlink throughput is set to 400Mbits/s (i.e., "iperf = 400 Mbit/s"), openhetnet0 (i.e., the virtual interface) = 49.80MBytes/s (captured by speedometer), eth0 = 26.60MBytes/s (captured by speedometer), and eth1 = 26.90 MBytes/s (captured by speedometer).

The red rectangle, dark blue rectangle, and pink rectangle in speedometer screen respectively highlights three observation windows which are related to openhetnet0, eth0, and eth1. From the speedometer screen, we can observe an interesting phenomenon that the sum of the two Ethernet interfaces (i.e., eth0 and eth1) theoretical downlink throughput is larger than the theoretical downlink throughput of openhetnet0. However, for generally speaking, the sum of the two Ethernet interfaces theoretical downlink throughput should be equals to the theoretical downlink throughput of openhetnet0, the reason contribute to this unexpected phenomenon is that in the virtual upper MAC module, each packet is added a virtual MAC header, therefore the size of the packet arrives at the Ethernet interface is larger than the size of the packet arrives at "openhetnet0". The red circles and blue circles in the right-side sub-figures are respectively refer to the actual downlink throughput in WiFi platform and in LTE platform.

The results in Fig.37, Fig.38, and Fig.39 show that when the theoretical downlink throughput lower than or equals to 400 Mbits/s (i.e., "iperf <= 400 Mbits/s"), the actual downlink throug-hphput lost ratio is approximately lower than 10%. Specificall-y, as shown in Fig.37, when the theoretical downlink throughp-uts of eth0 = 6.67MBytes/s and eth1 = 6.70Mbytes/s, the actua-l downlink throughputs of open WiFi platform are in the range of [5.9, 6.9] MBytes/s and the actual downlink throughputs of open LTE platform are in the range of [5.6, 6.6] MBytes/s. For calculation conveniently, we use the median of the range to cal-culate the throughput lost ratio incurred by the WiFi platform and by the LTE platform, then we can obtain the throughput lo-st ratio of open WiFi platform is (6.67-6.4)/6.67=0.04, and the throughput lost ratio of open LTE platform is (6.7-6.1)/6.7 = 0.09. Based on the same calculation procedures, in Fig. (38,39,40), we can respectively obtain the throughput lost ratios of ope-n WiFi platform are [0, (19.8-19.15)/19.8=0.033, (26.6-25.1)/26.6=0.056], and the throughput lost ratios of open LTE platfor-m are [(13.5-13)/13=0.03, (20-18.7)/20=0.065, (26.9-24.5)/26.9= 0.089].


root@jmakg4: /home/jmakg
Speedometer 2.8
63.4 MiB/s 62.8 MiB/s 63.3 MiB/s 63.4 MiB/s 63.4 MiB/s 62.8 MiB/s
TX: openhetnet0 0 B/s 0 B/s 10.8 MiB/s
33.8 MiB/s 33.8 MiB/s 33.8 MiB/s33.8 MiB/33.8 MiB/s 33.8 MiB/33.8 MiB/s
TX: eth1 0 B/s 0 B/s 5.76 MiB/s
33.6 MiB/s 33.6 MiB/s 33.6 MiB/s33.6 MiB/33.6 MiB/s33.6 MiB/33.6 MiB/s
TX: eth0 0 B/s 0 B/s 5.89 MiB/s

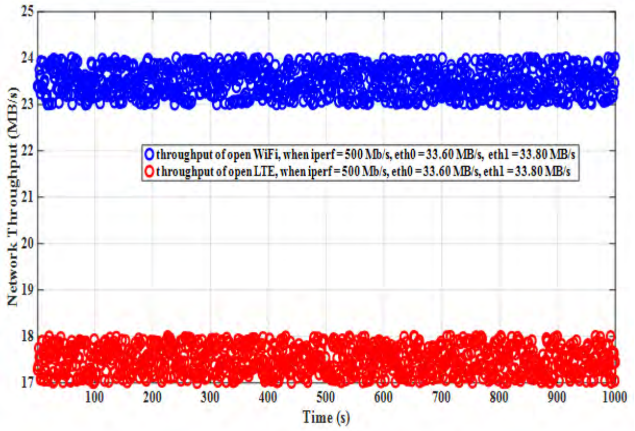
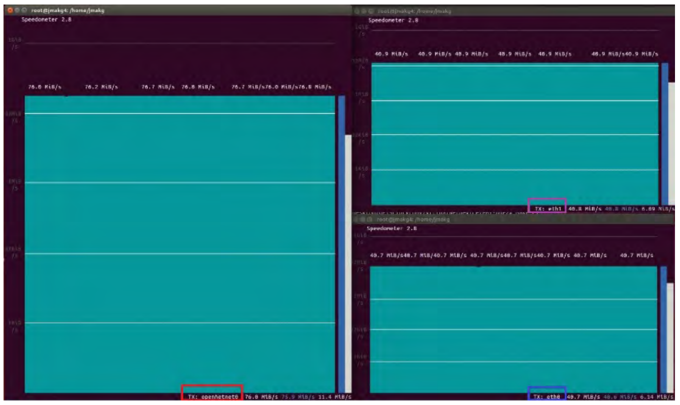


FIGURE 41. The actual throughput obtained by the open WiFi platform and the open LTE platform when the theoretical downlink throughput is set to 500Mbits/s (i.e., "iperf = 500 Mbit/s"), openhetnet0 (i.e., the virtual interface) = 63.10MBytes/s (captured by speedometer), eth0 = 33.60MBytes/s (captured by speedometer), and eth1 = 33.80 MBytes/s (captured by speedometer).


root@jmakg4: /home/jmakg
Speedometer 2.8
76.8 MiB/s 76.2 MiB/s 76.7 MiB/s 76.8 MiB/s 76.7 MiB/s76.0 MiB/s76.8 MiB/s
TX: openhetnet0 76.8 MiB/s 75.9 MiB/s 11.4 MiB/s
40.9 MiB/s 40.9 MiB/s 40.9 MiB/s 40.9 MiB/s 40.9 MiB/s 40.9 MiB/s40.9 MiB/s
TX: eth1 40.8 MiB/s 40.8 MiB/s 6.89 MiB/s
40.7 MiB/s40.7 MiB/40.7 MiB/s 40.7 MiB/s40.7 MiB/s40.7 MiB/s 40.7 MiB/s 40.7 MiB/s
TX: eth0 40.7 MiB/s 40.6 MiB/s 6.14 MiB/s

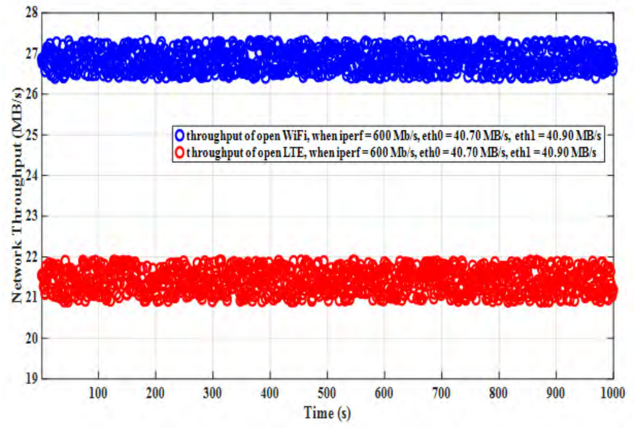


FIGURE 42. The actual throughput obtained by the open WiFi platform and the open LTE platform when the theoretical downlink throughput is set to 600Mbits/s (i.e., "iperf = 600 Mbit/s"), openhetnet0 (i.e., the virtual interface) = 76.60MBytes/s (captured by speedometer), eth0 = 40.70MBytes/s (captured by speedometer), and eth1 = 40.90 MBytes/s (captured by speedometer).

Then, from Fig.41 to Fig.46, we can easy observe that the actual downlink throughput lost ratio increases promptly with the theoretical downlink throughput increases. To be specific, thro-ugh the calculation, we can obtain the throughput lost ratios of open WiFi platform and open LTE platform in Fig.(41-46) are r-espectively equal to [(33.6-23.5)/33.6=0.3,(40.7-26.85)/40.7=0.34, (48.3-25.95)/48.3=0.46, (53.8-26.95)/53.8=0.49, (54-25.6)/54=0.53, (54.28-26.45)/54.28=0.51] and [(33.8-17.5)/33.8=0.48, (40.9-21.4)/40.9=0.048, (48.6-24.45)/48.6=0.5, (54-26.5)/54=0.51, (54.2-25.7)/54.2=0.53, (54.56-21.4)/54.56=0.61].

Based on the above-mentioned analyses, we can kindly conclude that the maximum actual downlink throughput which can be supported by our HetNets platform is approximately equals to 396Mbits/s.

IV. PACKETS OFFLOADING ANALYTICAL MODEL BASED ON SOFTWARE DEFINED OPEN HETNETS PLATFORM

In this section, based on the open HetNets platform described in previous section, we propose the packets offloading analytical model which is shown as below. The first state link in Fig.47 can be understood as the open WiFi platform shown in Fig.24 and the second state link can be thought as the open LTE platform.

As illustrated in previous section, the non-delayed single path traffic offloading (i.e., the first case shown in Fig.1) is a special case of the simultaneously dual path offloading. Therefore, in this section, we first propose the analytical model for the non-delayed single path offloading scheme and then propose the analytical model for the dual path offloading scheme.

A. THE NON-DELAYED SINGLE PATH OFFLOADING SCHEME

For analyzing conveniently, we model the non-delayed single path offloading scheme as a Markov chain shown in Fig.47. The symbols $SP_{j,CE}$ refers to the stationary probability of UE only in cellular coverage and there are $j$packets in the transmission queue (i.e., j-1waiting and one being transmitted over cellular network), and $SP_{j,WI}$ refers to the stationary probability of UE only in WiFi coverage and there are j packets in the transmission queue. In addition, the variables

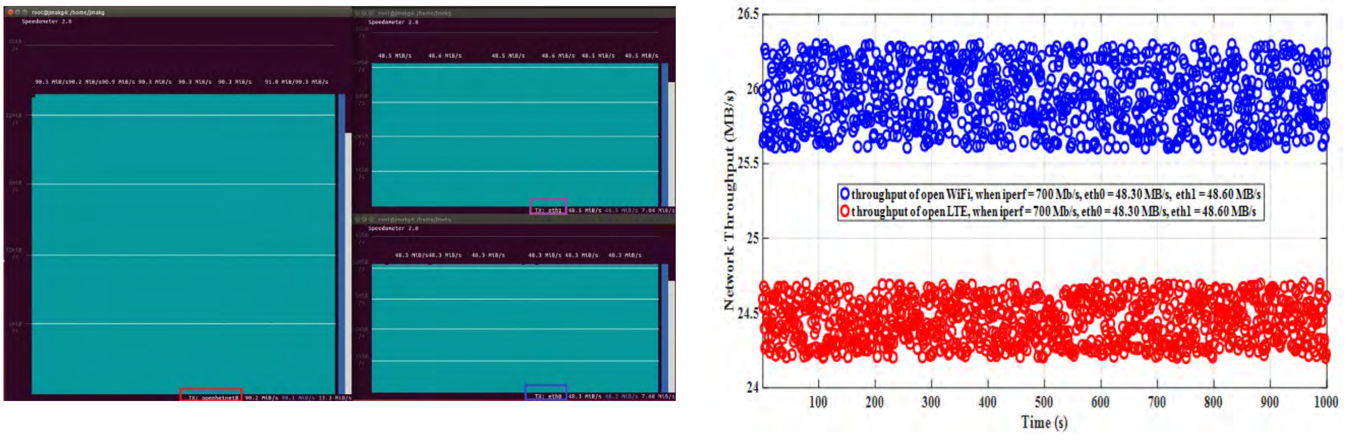


FIGURE 43. The actual throughput obtained by the open WiFi platform and the open LTE platform when the theoretical downlink throughput is set to 700Mbits/s (i.e., "iperf = 700 Mbit/s"), openhetnet0 (i.e., the virtual interface) = 90.20MBytes/s (captured by speedometer), eth0 = 48.30MBytes/s (captured by speedometer), and eth1 = 48.60 MBytes/s (captured by speedometer).

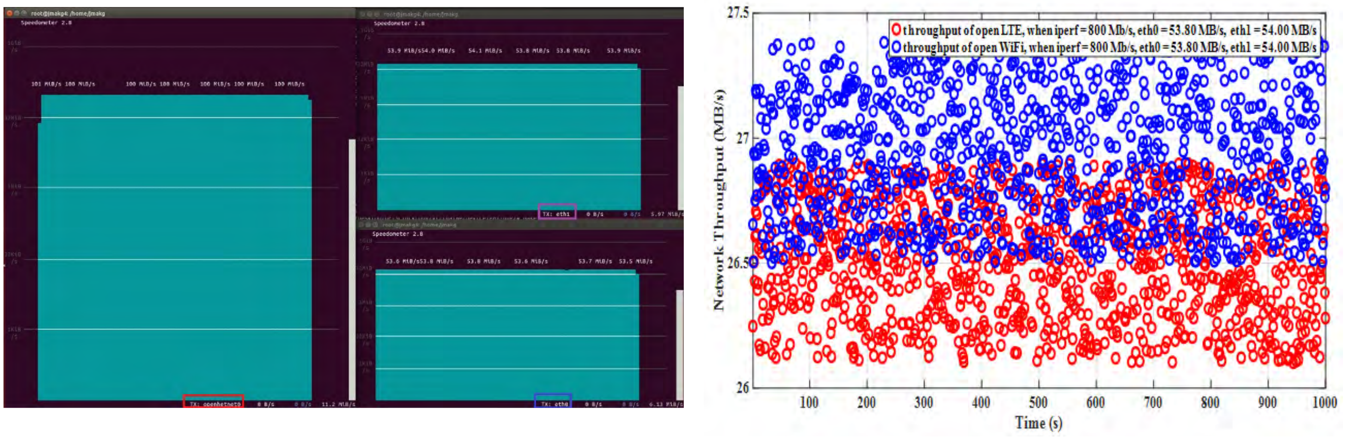


FIGURE 44. The actual throughput obtained by the open WiFi platform and the open LTE platform when the theoretical downlink throughput is set to 800Mbits/s (i.e., "iperf = 800 Mbit/s"), openhetnet0 (i.e., the virtual interface) = 100.00MBytes/s (captured by speedometer), eth0 = 53.80MBytes/s (captured by speedometer), and eth1 = 54.00 MBytes/s (captured by speedometer).

ξ_C refers to the packet Poisson arrival rate at UE in cellular network coverage and ξ_W is the packet Poisson arrival rate at UE in WiFi coverage. The variables θ_{CE} be the service rate while in cellular network coverage and θ_{WI} be the service rate while in WiFi coverage. Furthermore, the variables β_C denotes the rate of leaving the cellular state and β_W denotes the rate of leaving the WiFi state. Then, based on the variables notation and the Markov chain, we can obtain the balance equations shown below:

$$\begin{aligned}
j = 0:\ & SP_{0,CE}(\xi_C + \beta_C) = SP_{0,WI}\beta_W + SP_{1,CE}\theta_{CE}\\
& SP_{0,WI}(\xi_W + \beta_W) = SP_{1,WI}\theta_{WI} + SP_{0,CE}\beta_C\\
j > 0:\ & SP_{j,CE}(\xi_C + \beta_C + \theta_{CE})\\
& = SP_{j-1,CE}\xi_C + SP_{j+1,CE}\theta_{CE} + SP_{j,WI}\beta_W\\
& SP_{j,WI}(\xi_W + \beta_W + \theta_{WI})\\
& = SP_{j-1,WI}\xi_W + SP_{j+1,WI}\theta_{WI} + SP_{j,CE}\beta_C \qquad (3)
\end{aligned}$$

Based on the above-defined stationary probability (i.e., $SP_{j,CE}$ and $SP_{j,WI}$) and reference [27], we then give the probability generating functions (PGF), which are shown below, for the WiFi state and cellular state.

$$\begin{aligned}
G_{CE}(Z) &= \sum_{j=0}^{\infty} SP_{j,CE}Z^j\\
G_{WI}(Z) &= \sum_{j=0}^{\infty} SP_{j,WI}Z^j\\
|Z| &\le 1 \qquad (4)
\end{aligned}$$

We rewrite the first sub-equation in equation (3) and multiply Z^0, then we can get equation (5):

$$\begin{aligned}
& SP_{0,CE}(\xi_C + \beta_C + \theta_{CE})\\
&\quad = SP_{0,WI}\beta_W + SP_{1,CE}\theta_{CE} + SP_{0,CE}\theta_{CE}\\
& SP_{0,CE}Z^0(\xi_C + \beta_C + \theta_{CE})\\
&\quad = SP_{0,WI}\beta_W Z^0 + SP_{1,CE}\theta_{CE}Z^0 + SP_{0,CE}\theta_{CE}Z^0 \qquad (5)
\end{aligned}$$

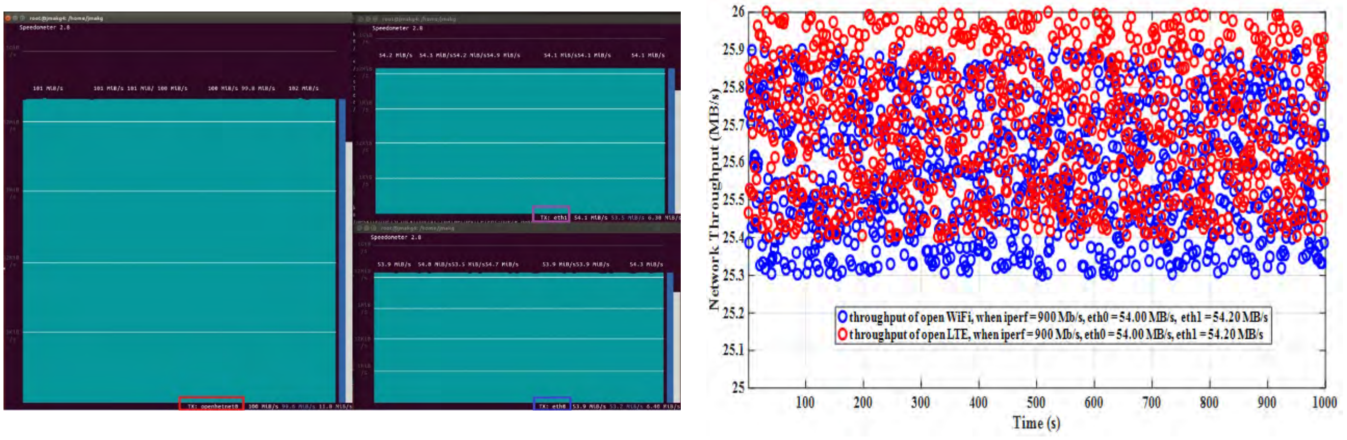


FIGURE 45. The actual throughput obtained by the open WiFi platform and the open LTE platform when the theoretical downlink throughput is set to 900Mbits/s (i.e., "iperf = 900 Mbit/s"), openhetnet0 (i.e., the virtual interface) = 101.00MBytes/s (captured by speedometer), eth0 = 54.00MBytes/s (captured by speedometer), and eth1 = 54.20 MBytes/s (captured by speedometer).

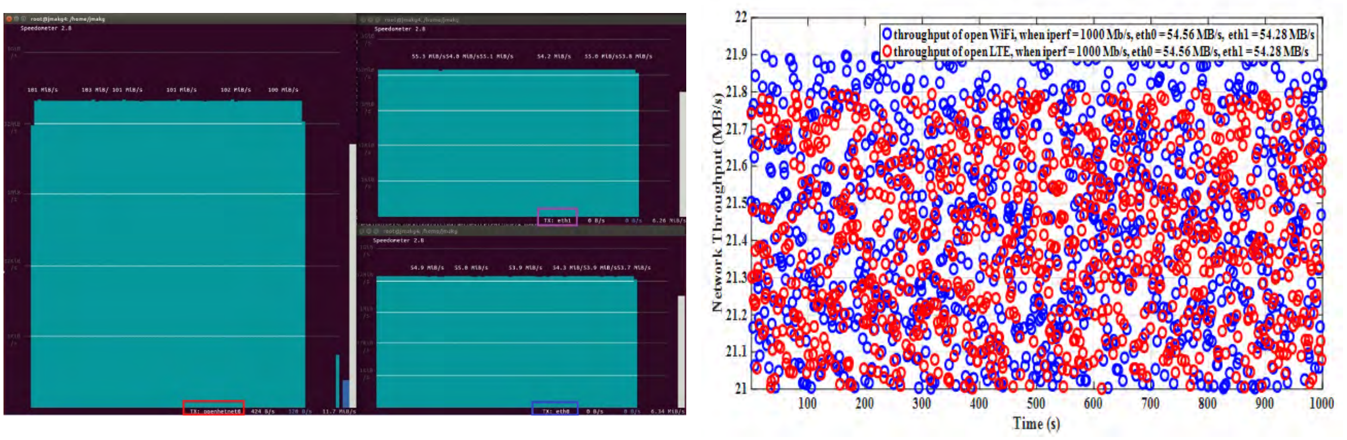


FIGURE 46. The actual throughput obtained by the open WiFi platform and the open LTE platform when the theoretical downlink throughput is set to 1000Mbits/s (i.e., "iperf = 1000 Mbit/s"), openhetnet0 (i.e., the virtual interface) = 102.00MBytes/s (captured by speedometer), eth0 = 54.56MBytes/s (captured by speedometer), and eth1 = 54.28 MBytes/s (captured by speedometer).

In the same way, we rewrite the third sub-equation in equation (3) and multiply Z^j, and then we can obtain equation (6):

$$\begin{aligned}
&SP_{j,CE}(\xi_C + \beta_C + \theta_{CE}) \\
&\quad = SP_{j-1,CE}\xi_C + SP_{j,WI}\beta_W + SP_{j+1,CE}\theta_{CE} \\
&SP_{j,CE}Z^j(\xi_C + \beta_C + \theta_{CE}) \\
&\quad = SP_{j-1,CE}\xi_C Z^j + SP_{j,WI}\beta_W Z^j + SP_{j+1,CE}\theta_{CE}Z^j \\
&j > 0
\end{aligned} \tag{6}$$

Based on the second sub-equation in equation (5) and the second sub-equation in equation (6), we sum all j's (i.e., $j = 1, 2, 3, \ldots$). After some derivations we get the following equation (7), the detailed derivation process is illustrated in Appendix A. The symbol SE in equation (7) is the abbreviation of the word "sub-equation".

$$\begin{aligned}
(\xi_C + \beta_C + \theta_{CE})G_{CE}(Z) &= SE_1 + SE_2 + SE_3 + SE_4 \\
SE_1 &= \xi_C Z G_{CE}(Z) \\
SE_2 &= \beta_W G_{WI}(Z) \\
SE_3 &= \frac{\theta_{CE}}{Z}(G_{CE}(Z) - SP_{0,CE}) \\
SE_4 &= SP_{0,CE}\theta_{CE}
\end{aligned} \tag{7}$$

Repeating the same process for the second sub-equation in equation (3) and the fourth sub-equation in equation (3), we can obtain equation (8) (i.e., for WiFi state):

$$\begin{aligned}
(\xi_W + \beta_W + \theta_{WI})G_{WI}(Z) &= SE_1 + SE_2 + SE_3 + SE_4 \\
SE_1 &= \xi_W Z G_{WI}(Z) \\
SE_2 &= \beta_C G_{CE}(Z) \\
SE_3 &= \frac{\theta_{WI}}{Z}(G_{WI}(Z) - SP_{0,WI}) \\
SE_4 &= SP_{0,WI}\theta_{WI}
\end{aligned} \tag{8}$$

Note that equation (7) and equation (8) define a system of equations in $G_{CE}(Z)$ and $G_{WI}(Z)$, from which we can get the

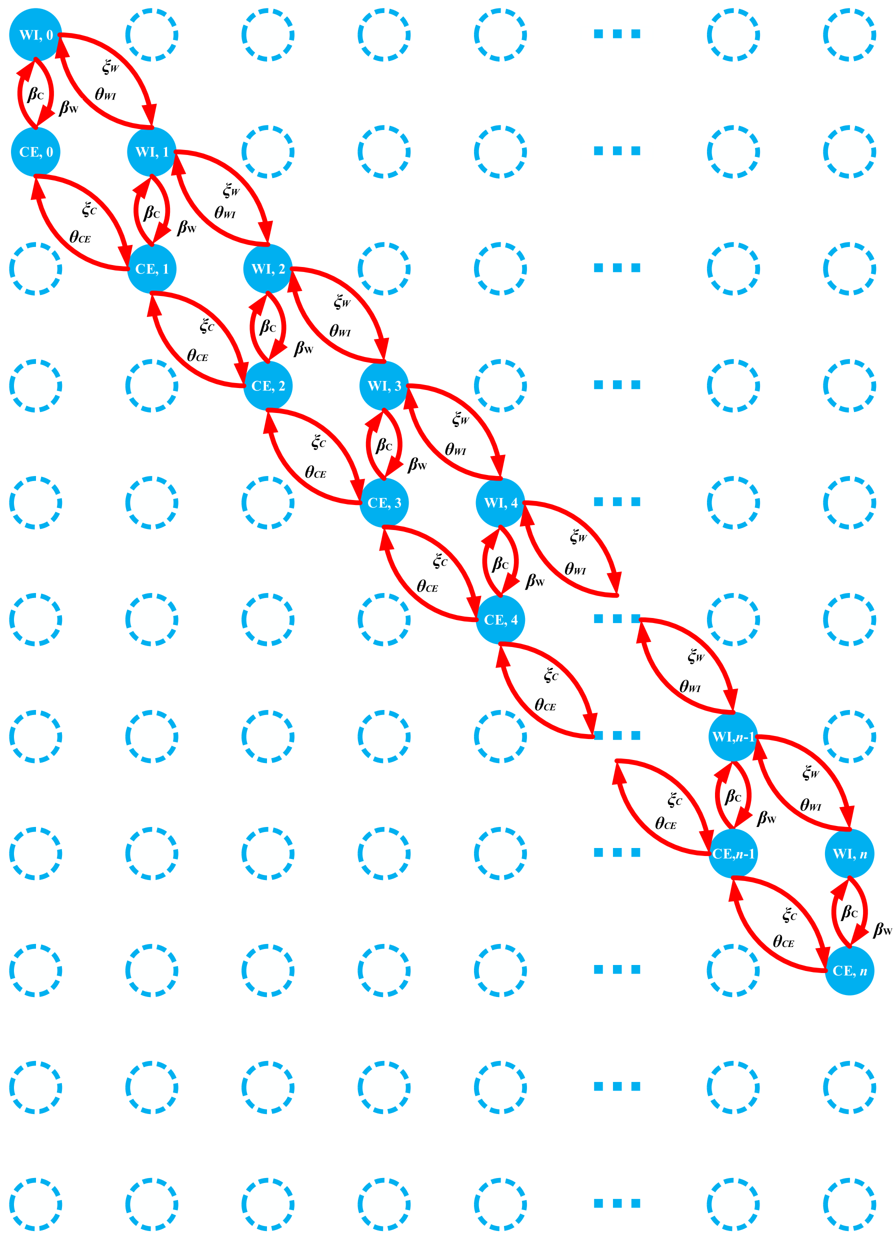


FIGURE 47. The 2-dimensional Markov chain for non-delayed single path offloading scheme.

following equation (9):

$$\begin{aligned} f(Z)G_{CE}(Z) &= SE_1 + SE_2 \\ SE_1 &= SP_{0,WI}\beta_W\theta_{WI}Z \\ SE_2 &= SP_{0,CE}\theta_{CE}[\beta_W Z \\ &\quad +\xi_W Z(1-Z) - \theta_{WI}(1-Z)] \end{aligned} \tag{9}$$

The function $f(Z)$ in equation (9) can be replaced by equation (10).

$$\begin{aligned} f(Z) &= SE_1 + SE_2 + SE_3 + SE_4 \\ SE_1 &= \xi_C\xi_W Z^3 \\ SE_2 &= -(\beta_C\xi_W + \beta_W\xi_C + \xi_C\xi_W + \xi_C\theta_{WI} + \xi_W\theta_{CE})Z^2 \\ SE_3 &= (\beta_C\theta_{WI} + \beta_W\theta_{CE} + \theta_{CE}\theta_{WI} + \xi_C\theta_{WI} + \xi_W\theta_{CE})Z \\ SE_4 &= -\theta_{CE}\theta_{WI} \end{aligned} \tag{10}$$

Yechiali and Naor [27] have proven that the polynomial in equation (10) has only one root in the open interval (0,1). Here, we denote this root as $Z_0$and set Z = Z_0into equation (10), then we can obtain equation (11):

$$\begin{aligned} &SE_1+SE_2=0 \\ &\quad SE_1=SP_{0,WI}\beta_W\theta_{WI}Z_0 \\ &\quad SE_2=SP_{0,CE}\theta_{CE}[\beta_W Z_0+\xi_W Z_0(1-Z_0)-\theta_{WI}(1-Z_0)] \end{aligned} \tag{11}$$

Based on Fig.47, we can obtain another balance equation by a vertical cut between states containing j and $j+1$ packet.

$$\begin{aligned} &SP_{j,CE}\xi_C + SP_{j,WI}\xi_W \\ &\quad = SP_{j+1,CE}\theta_{CE} + SP_{j+1,WI}\theta_{WI} \quad (j=0,1,2,\cdots\infty) \end{aligned} \tag{12}$$

Then we sum (12) over all j's (i.e., from $j = 1$ to ∞), we can obtain

$$\begin{aligned} &\xi SP_{\cdot,CE}\xi_C + SP_{\cdot,WI}\xi_W \\ &\quad = (SP_{\cdot,CE} - SP_{0,CE})\theta_{CE} + (SP_{\cdot,WI} - SP_{0,WI})\theta_{WI} \\ SP_{\cdot,CE} &= \sum_{j=0}^{\infty} SP_{j,CE}, \; SP_{\cdot,WI} = \sum_{j=0}^{\infty} SP_{j,WI} \end{aligned} \tag{13}$$

Then, let two variables θ_{AVE} and ξ_{AVE} be defined as:

$$\begin{aligned} \xi_{AVE} &= SP_{\cdot,CE}\xi_C + SP_{\cdot,WI}\xi_W \\ \theta_{AVE} &= SP_{\cdot,CE}\theta_{CE} + SP_{\cdot,WI}\theta_{WI} \end{aligned} \tag{14}$$

Based on equation (14), equation (13) can be rewritten as equation (15), the detailed derivation process from equation (13) to equation (15) is illustrated in Appendix B.

$$SP_{0,CE}\theta_{CE} + SP_{0,WI}\theta_{WI} = \theta_{AVE} - \xi_{AVE} \tag{15}$$

Then, based on equation (11) and equation (15), after some calculations, we can obtain equation (16). The detailed derivation process from equation (11) and equation (15) to equation (16) is illustrated in Appendix C.

$$\begin{aligned} SP_{0,CE} &= \frac{\beta_W(\theta_{AVE} - \xi_{AVE})Z_0}{\theta_{CE}(1 - Z_0)(\theta_{WI} - \xi_W Z_0)} \\ SP_{0,WI} &= \frac{\beta_C(\theta_{AVE} - \xi_{AVE})Z_0}{\theta_{WI}(1 - Z_0)(\theta_{CE} - \xi_C Z_0)} \end{aligned} \tag{16}$$

In addition, based on equation (9) and equation (15), we can further obtain:

$$\begin{aligned} &G_{CE}(Z) \\ &= \frac{[\beta_W(\theta_{AVE} - \xi_{AVE})Z + SP_{0,CE}\theta_{CE}(1 - Z)(\xi_W Z - \theta_{WI})]}{f(Z)} \\ &G_{WI}(Z) \\ &= \frac{[\beta_C(\theta_{AVE} - \xi_{AVE})Z + SP_{0,WI}\theta_{WI}(1 - Z)(\xi_C Z - \theta_{CE})]}{f(Z)} \end{aligned} \tag{17}$$

We then define two new variables E[N_{WI}] and E[N_{CE}] shown in equation (18), where the E[N_{WI}] refers to the average number of packets will be transmitted by WiFi and E[N_{CE}] refers to the average number of packets will be transmitted by LTE (i.e., the cellular network).

$$E[N_{WI}] = \sum_{j=0}^{\infty} jSP_{j,WI}, \; E[N_{CE}] = \sum_{j=0}^{\infty} jSP_{j,CE} \tag{18}$$

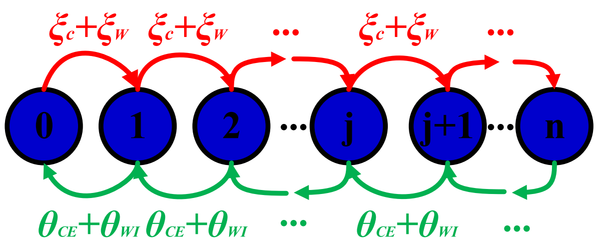

$\xi_c+\xi_w$ $\xi_c+\xi_w$ ••• $\xi_c+\xi_w$ •••
0
1
2
j
j+1
n
$\theta_{CE}+\theta_{WI}$ $\theta_{CE}+\theta_{WI}$ ••• $\theta_{CE}+\theta_{WI}$ •••

FIGURE 48. The 1-dimensional Markov chain for simultaneous dual path offloading scheme.

Let's go further step, we can find equation (18) has the following relationship with equation (19).

$$\begin{aligned} G'_{CE}(Z)|_{Z=1} &= E[N_{CE}] = \sum_{j=0}^{\infty} jSP_{j,CE} \\ G'_{WI}(Z)|_{Z=1} &= E[N_{WI}] = \sum_{j=0}^{\infty} jSP_{j,WI} \end{aligned} \tag{19}$$

Then, it is easy to conclude that the average number of packets in UE is:

$$E[N] = E[N_{CE}] + E[N_{WI}] \tag{20}$$

We then insert $Z = 1$ in equation (17) and sum up those two sub-equations, we can get the average number of packets in system is (21), as shown at the bottom of this page.

Then, based on the Little's law E[N] = ξ_{AVE}E[T] [28], we can obtain average packet transmission delay in non-delayed single path (i.e., special case for dual path) mobile data offloading is (22), as shown at the bottom of this page.

B. THE SIMULTANEOUS DUAL PATH OFFLOADING SCHEME

For the simultaneous dual path offloading scheme, we utilize the 1-dimensional Markov chain shown in Fig.48 to analyze it. The dark blue circle refers to the number of packets arrived in UE is$j$ ($0 < j < n$)), with one packet being transmitted by WiFi and the other packet being transmitted by LTE. The red arrow line denotes the packet Poisson arrival rate at UE in overlaid area (i.e., the overlaid area shown in Fig.48, which is composed by WiFi and LTE) is $\xi_W + \xi_C$. The green arrow line denotes the service rate in the overlaid area is $\theta_{CE} + \theta_{WI}$.

$$\begin{aligned} E[N] &= E[N_{CE}] + E[N_{WI}] \\ &= \frac{\xi_{AVE}}{\theta_{AVE} - \xi_{AVE}} + \frac{[\theta_{CE}(\theta_{WI} - \xi_W)SP_{0,CE} + \theta_{WI}(\theta_{CE} - \xi_C)SP_{0,WI} - (\theta_{CE} - \xi_C)(\theta_{WI} - \xi_W)]}{(\beta_C + \beta_W)(\theta_{AVE} - \xi_{AVE})} \end{aligned} \tag{21}$$

$$E[T] = \frac{1}{\theta_{AVE} - \xi_{AVE}} + \frac{[\theta_{CE}(\theta_{WI} - \xi_W)SP_{0,CE} + \theta_{WI}(\theta_{CE} - \xi_C)SP_{0,WI} - (\theta_{CE} - \xi_C)(\theta_{WI} - \xi_W)]}{(\beta_C + \beta_W)(\theta_{AVE} - \xi_{AVE})\xi_{AVE}} \tag{22}$$

Based on Fig.48, we can obtain the balance equation for this 1-dimensional Markov chain is:

$$\begin{cases} SP_0(\xi_w + \xi_c) = SP_1(\theta_{Wi} + \theta_{CE}) & (j = 0) \\ SP_{j-1}(\xi_w + \xi_c) = SP_j(\theta_{Wi} + \theta_{CE}) & (0 \le j \le n, m = \pm\infty) \end{cases} \tag{23}$$

It is obviously equation (24) is true.

$$\sum_{j=0}^{n} SP_j = 1 \quad (n = +\infty) \tag{24}$$

Then, for derivation conveniently, let us define a variable k shown in equation (25), which denotes the ratio between the packets arrival rate and the packets service rate.

$$k = \frac{\xi_W + \xi_C}{\theta_{Wi} + \theta_{CE}} \tag{25}$$

Based on the above-mentioned analysis, we can obtain the average number of packets in the system is:

$$\begin{aligned} E(N) &= \sum_{j=0}^{n} SP_j \quad (n = +\infty) \\ &= (k + 2k^2 + 3k^2 + \cdots) - (k^2 + 2k^3 + 3k^4 + \cdots) \\ &= k + k^2 + k^3 + \cdots \\ &= \frac{k}{1-k} = \frac{\xi_C + \xi_W}{\theta_{WI} + \theta_{CE} - \xi_C - \xi_W} \end{aligned} \tag{26}$$

Based on the Little's law $\text{E}[N] = \xi_C + \xi_W \ \text{E}[T]$, we can obtain average packet transmission delay in simultaneous dual path mobile data offloading is:

$$\begin{aligned} E(T) &= \frac{1}{\xi_C + \xi_W}(E(N)) \\ &= \frac{1}{\theta_{WI} + \theta_{CE} - \xi_C - \xi_W} \end{aligned} \tag{27}$$

C. THE OFFLOADING EFFICIENCY

Finally, an important parameter that can be quantitatively characterize data offloading is the offloading efficiency OE, defined as the ratio of the amount of transmitted data through WiFi interface against the total amount of transmitted data. The higher offloading efficiency means better performance for both the UE and the network side. Therefore, knowing this parameter is especially important when it comes to calculating how much a UE will have to pay, knowing that the charges for using Internet access are not the same for WiFi as are for cellular network.

Let $t_w(t_c)$ refers to the total time during which data are transmitted through the WiFi (cellular) interface. Then, we can define the offloading efficiency to be the percentage of data transmitted through the WiFi network, we have the following expression for it:

$$OE = \frac{\theta_{WI} t_{WI}}{\theta_{CE} t_C + \theta_{WI} t_{WI}} \tag{28}$$

V. EXPERIMENTATION AND EVALUATION

In this section, based on the software defined open HetNets platform, we carry out the packets offloading experiment to verify the analytical results (i.e., the transmission delay and the offloading efficiency) illustrated in section IV. Specifically, we mainly focus on the impact of the packets arrival rate in the soft UE side, the off-period of open WiFi platform, the system utilization of open WiFi platform, and the system utilization of open LTE platform for the offloading performance. An experiment of such scenario afore-mentioned is illustrated in Fig.49 and the detailed experiment parameters are shown in Table I. Note that in Fig.49, each software-defined device (i.e., the software-defined UE, the software-defined open WiFi platform, and the software-defined open LTE platform) is composed by one NUC and one USRP. Here, we let the soft UE as the transmitter and let the open HetNets platform as the receiver. Then, based on the experiment plan and the experiment parameters, we conduct three group experiments. In each group experiment, we first keep the packets arrival rate in soft UE unchanged, the system utilization of open WiFi platform unchanged, and the system utilization of open LTE platform unchanged, and then change the off-period of open WiFi platform to evaluate the main performance metrics including the average transmission delay and the offloading efficiency. To be specific, in section V.A, we present the first group experiment and the evaluation process. Then, in section V.B we give the description for the second group experiment and evaluation process. Finally, in section V.C, we conduct the third group experiment and make some evaluation comments.

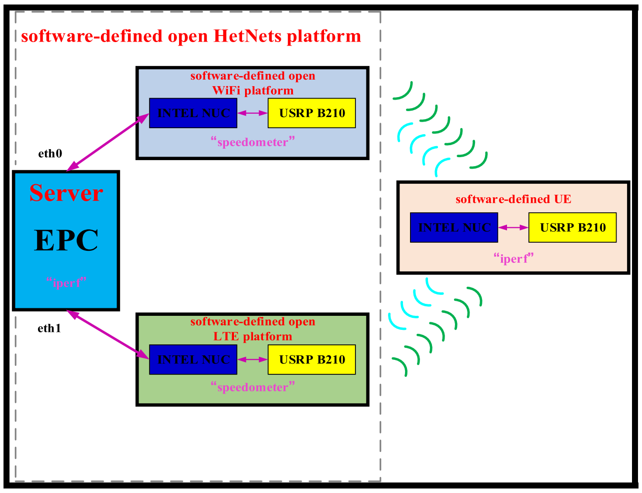


FIGURE 49. The experiment plan for performance testing.

A. PERFORMANCE UNDER THE FIRST GROUP EXPERIMENT PARAMETER SETTING

In this sub-section, we conduct the first group experiment to evaluate the analytical results proposed in section IV. Specifically, we carry out the following experiment procedures.

(1). We deploy three software defined devices and one server in our laboratory. Each software device is composed by the INTEL NUC and USRP.

(2). In the server side, we implement ten modules (i.e., the enhanced EPC) into it, which includes the virtual upper MAC module, the virtual lower MAC module (i.e., the Offloading Engine module), and eight virtual PHY modules (i.e., eight Offloading Observing Window modules).

(3). For those three software defined devices including software defined UE, software defined open WiFi platform and software defined open LTE platform, we respectively compile and run them.

(4). In the server side, we open a command window and type the command "iperf –s –u –i" to set it to the server mode. The iperf is a network performance testing tool, which can test the maximum uplink (i.e., from client to server) TCP/UDP bandwidth, the maximum downlink (i.e., from server to client) TCP/UDP bandwidth, the average transmission delay, and etc. Here, we give the explanation of each parameter in this command. The first parameter "-s" indicates that this server is configured to the server mode. The second parameter "-u" indicates that transmitter must use the UDP session to transmit packets to this server. The third parameter "-i" indicates that the time interval of outputting the testing report in the server side is 1 second. Similarly, in the soft UE side, we open a command window and type the command "iperf -c -u 192.168.9.1 -i 1 -t 1800 -b 100M". We give the explanation of each parameter in this command. The first parameter "-c" indicates that the soft UE is configured to the client mode. The second parameter "-u" indicate that this UE use the UDP session to transmit packets to this server. The third parameter "-i" indicates that the time interval of outputting testing report in the UE side is 1 second. The fourth parameter "-t 1800" indicates that the total observing time interval is 1800 seconds (i.e., 30 minutes). The fifth parameter "-b 100M indicates that the theoretical uplink bandwidth (i.e., from client to server) is set to 100 Mbps.

(5). Based on the above-mentioned procedures, in the open WiFi platform and open LTE platform, we respectively run the network speed testing tool "speedometer" (i.e., the pink words shown in Fig.49) to observe the obtained throughput in those two platforms over time.

(6). In order to observe the opportunistic single-path packets transmission performance, we keep the packets arrival rate in UE side unchanged and keep the system utilization of open WiFi platform and open LTE platform unchanged, we change the off period of open WiFi platform. For example, we turn off the open WiFi platform in the twenty-fifth minutes (i.e., set the off period of open WiFi platform to 5 minutes). In another words, from twenty-fifth minutes to thirty minutes, there is only one path to transmit the on-the-fly packets received from the soft UE (i.e., only the LTE path and no WiFi offloading assistance). Moreover, we utilize the thread mechanism to control the system utilization in the open WiFi platform and in the open LTE platform. To be specific, we can increase the system utilization by turning on some threads, we can decrease the system utilization by turning off some threads, and we can observe the system utilization information by inputting the command "top -c" in Linux system.

(7). In each minute, we gather the experiment reports including the average transmission delay and the practical throughput, and calculate the offloading efficiency (i.e., the total transmitted data through the open WiFi platform divide by the total transmitted data through the open WiFi platform and the open LTE platform). Based on the experiment results, we draw the following three statistic charts which are shown from Fig.50-Fig.52.

1) AVERAGE TRANSMISSION DELAY

Fig.50 (a), Fig.51 (a), and Fig.52 (a) demonstrate the average transmission delay under the first group experiment parameter setting (i.e., the packets arrival rate in UE side is set to 100Mbits/s, the execution time of "iperf" = 30 minute, the system utilization of open WiFi platform = 10%, and the system utilization of open LTE platform = 10%), and under the condition that the off period of open WiFi platform change from 5 minutes to 15 minutes. We can see the average transmission delay of the single one path (i.e., during the WiFi off period and there is only the LTE path exists) is always larger than the dual path (both the WiFi path and the LTE path exist) with the reason that offloading some data traffic to WiFi platform can relieve the burden of the LTE platform thus it can reduce the average transmission delay. For example, as shown in Fig.50(a), when the "iperf" execution time period locate between the first minute and the twenty-fifth minute (i.e., the dual path), the average transmission delay is lower than 2s; when the time period locate between the twenty-fifth minute and the thirtieth minute (i.e., the single path), the average transmission delay is larger than 2s. For Fig.51(a) and Fig.52(a), we can see the same trend as that shown in Fig.50(a). Besides, we can observe that the theoretical transmission delay is roughly aligned with experiment results in all considered cases. The reason for some deviations between the theoretical and experiment results is due to the fact that in the analytical model we always adopt some assumptions to derive the theoretical results. Therefore, when it comes to the real experiment results, the deviations are inevitable.

2) THROUGHPUT OBTAINED IN OPEN WIFI PLATFORM

Fig.50 (b), Fig.51 (b), and Fig.52 (b) depict the throughput obtained in the open WiFi platform under the first group experiment parameter setting, and under the condition that the off period of open WiFi platform change from 5 minutes to 15 minutes. From those three figures, we can observe that when in the dual path offloading state, the actual throughput obtained in the open WiFi platform is approximately equal to 80 Mbits/s. Since we have set the theoretical uplink throughput in the soft UE side to100 Mbits/s, therefore, it is easy to estimate the actual throughput obtained in the open

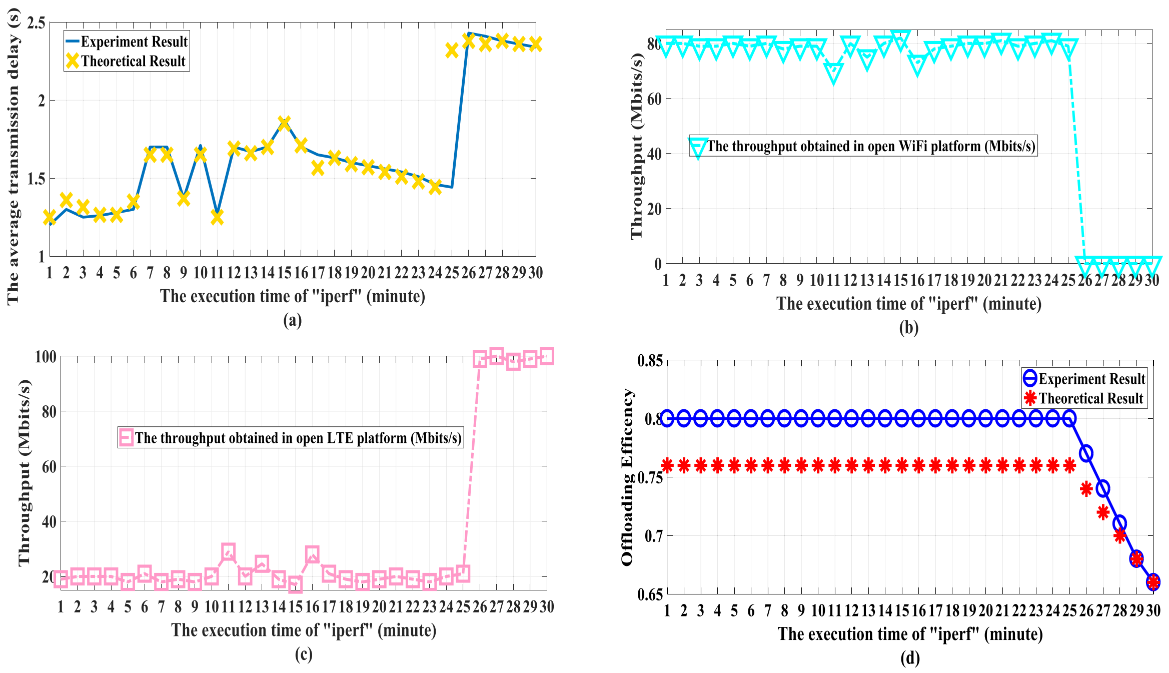


FIGURE 50. Experiment (solid lines) and theoretical (markers) results when the packets arrival rate in UE side is 100Mbits/s (i.e., "iperf" = 100Mbits/s), the execution time of "iperf" = 30 minute, the system utilization of open WiFi platform and open LTE platform = 10%, and the off period of open WiFi platform = 5 minute. (a) The average transmission delay; (b) The throughput obtained in the open WiFi platform; (c) The throughput obtained in the open LTE platform; (d) The offloading efficiency.

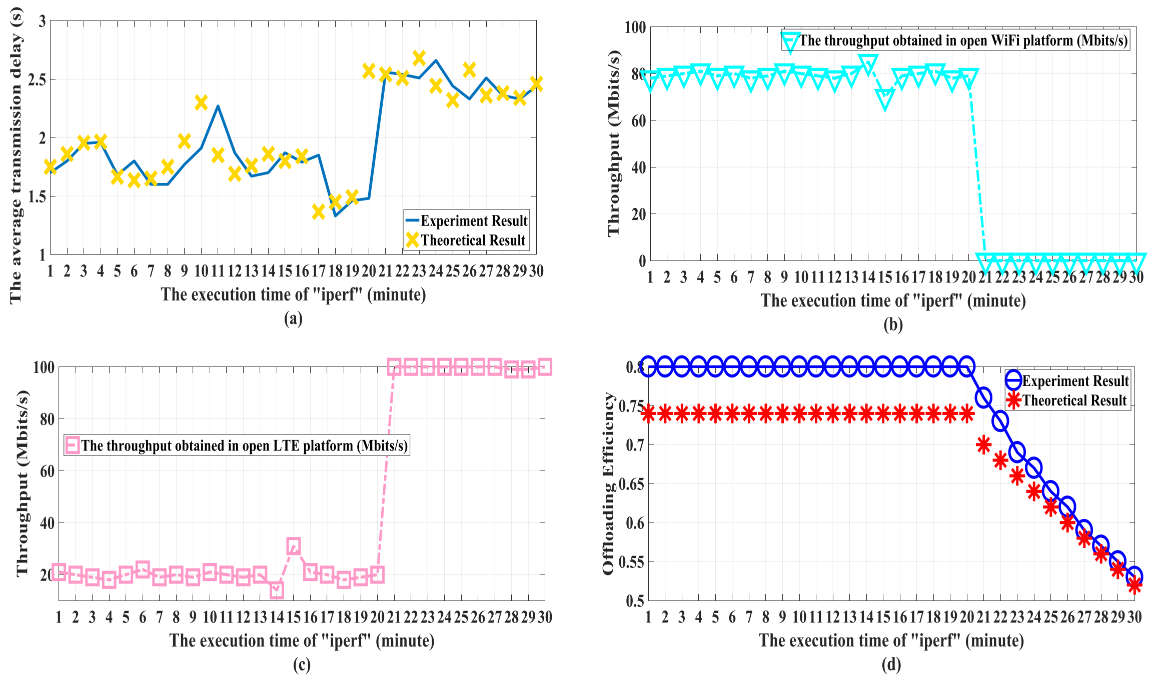


FIGURE 51. Experiment (solid lines) and theoretical (markers) results when the packets arrival rate in UE side is 100Mbits/s (i.e., "iperf" = 100Mbits/s), the execution time of "iperf" = 30 minute, the system utilization of open WiFi platform and open LTE platform = 10%, and the off period of open WiFi platform = 10 minute. (a) The average transmission delay; (b) The throughput obtained in the open WiFi platform; (c) The throughput obtained in the open LTE platform; (d) The offloading efficiency.

LTE platform is approximately equal to 20 Mbits/s, and the experiment results shown in Fig.50(c)-Fig.52(c) have proven what we have analyzed. Beside the dual path offloading state, when in the single one path state (i.e., during the open WiFi platform off period), the actual throughput in the WiFi platform is equal to 0Mbits/s and the actual throughput in

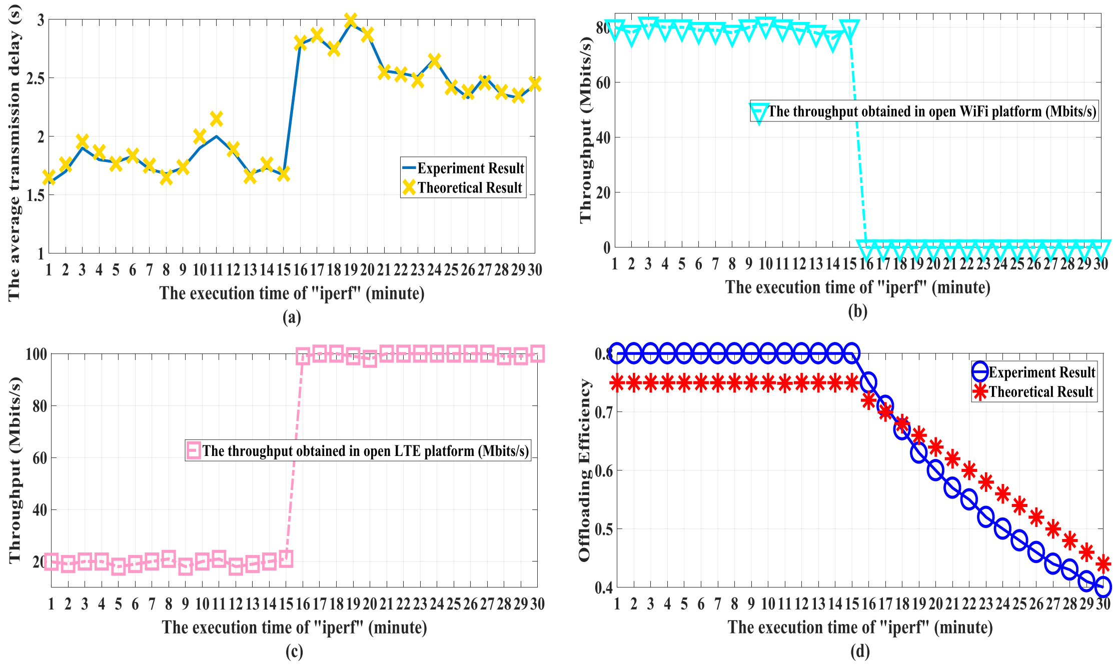


FIGURE 52. Experiment (solid lines) and theoretical (markers) results when the packets arrival rate in UE side is 100Mbits/s (i.e., "iperf" = 100Mbits/s), the execution time of "iperf" = 30 minute, the system utilization of open WiFi platform and open LTE platform = 10%, and the off period of open WiFi platform = 15 minute. (a) The average transmission delay; (b) The throughput obtained in the open WiFi platform; (c) The throughput obtained in the open LTE platform; (d) The offloading efficiency.

the LTE platform almost equals to 100Mbits/s, which bring a great burden to the cellular platform.

3) THROUGHPUT OBTAINED IN OPEN LTE PLATFORM

Fig.50 (c), Fig.51 (c), and Fig.52 (c) illustrate the throughput obtained in the open LTE platform under the first group experiment parameter setting, and under the condition that the off period of open WiFi platform change from 5 minutes to 15 minutes. Based on those three figures, we can see that when in the dual path offloading state, the actual throughput obtained in the open LTE platform is approximately equal to 20 Mbits/s, this experiment results are corresponding to the results shown in Fig.50(b)-Fig.52(b) (i.e., the sum is 100 Mbits/s). Moreover, as described above, when in the single one path state (i.e., during the open WiFi platform off period), the actual throughput in the open LTE platform almost equals to 100Mbits/s,

4) OFFLOADING EFFICIENCY

Fig.50 (d), Fig.51 (d), and Fig.52 (d) demonstrate the offloading efficiency under the first group experiment parameter setting, and under the condition that the off period of open WiFi platform change from 5 minutes to 15 minutes. Note that the offloading efficiency of the dual path (i.e., both the WiFi path and the LTE path exist) is always larger than the single one path (i.e., during the WiFi off period and there is only the LTE path exists) with the reason that the offloading efficiency is calculated based on the total packets transmitted through the WiFi platform divided by the total packets transmitted through the WiFi platform and the open LTE platform. For instance, as illustrated in Fig.52(d), when the "iperf" execution time period locate between the first minute and the fifteenth minute (i.e., the dual path), the offloading efficiency is equal to 0.8; when the time period locate between the sixteenth minute and the thirtieth minute (i.e., the single path), the offloading efficiency is lower than 0.7. For Fig.50 (d) and Fig.51 (d), we can observe the same trend as that shown in Fig.52 (d). Moreover, we can observe that the theoretical transmission delay is roughly aligned with experiment results in all considered cases. The reason for some deviations between the theoretical and experiment results is due to the fact that the performance of open WiFi platform and open LTE platform are still need to be further improved.

B. PERFORMANCE UNDER THE SECOND GROUP EXPERIMENT PARAMETER SETTING

In this sub-section, we carry out the second group experiment to evaluate the analytical results proposed in section IV. The experiment procedures are the same as that described in section V.A. Based on the second group experiment results, we draw the following three statistic graphs which are shown from Fig.53-Fig.55.

1) AVERAGE TRANSMISSION DELAY

Fig.53 (a), Fig.54 (a), and Fig.55 (a) demonstrate the average transmission delay under the second group experiment

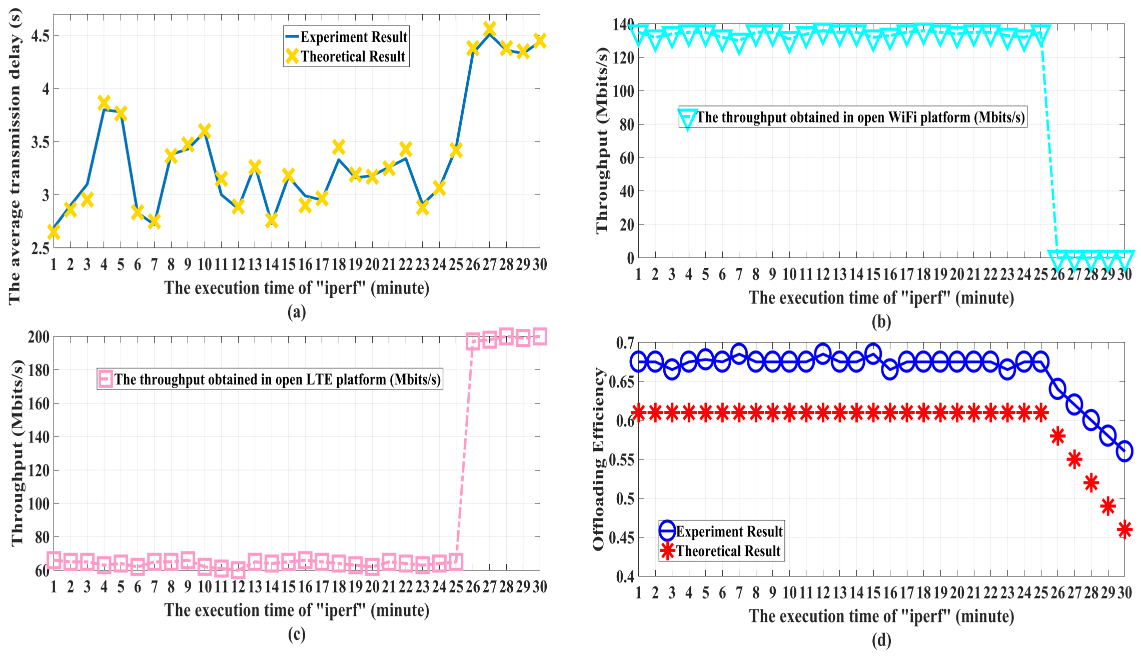


FIGURE 53. Experiment (solid lines) and theoretical (markers) results when the packets arrival rate in UE side is 200Mbits/s (i.e., "iperf" = 200Mbits/s), the execution time of "iperf" = 30 minute, the system utilization of open WiFi platform and open LTE platform = 15%, and the off period of open WiFi platform = 5 minute. (a) The average transmission delay; (b) The throughput obtained in the open WiFi platform; (c) The throughput obtained in the open LTE platform; (d) The offloading efficiency.

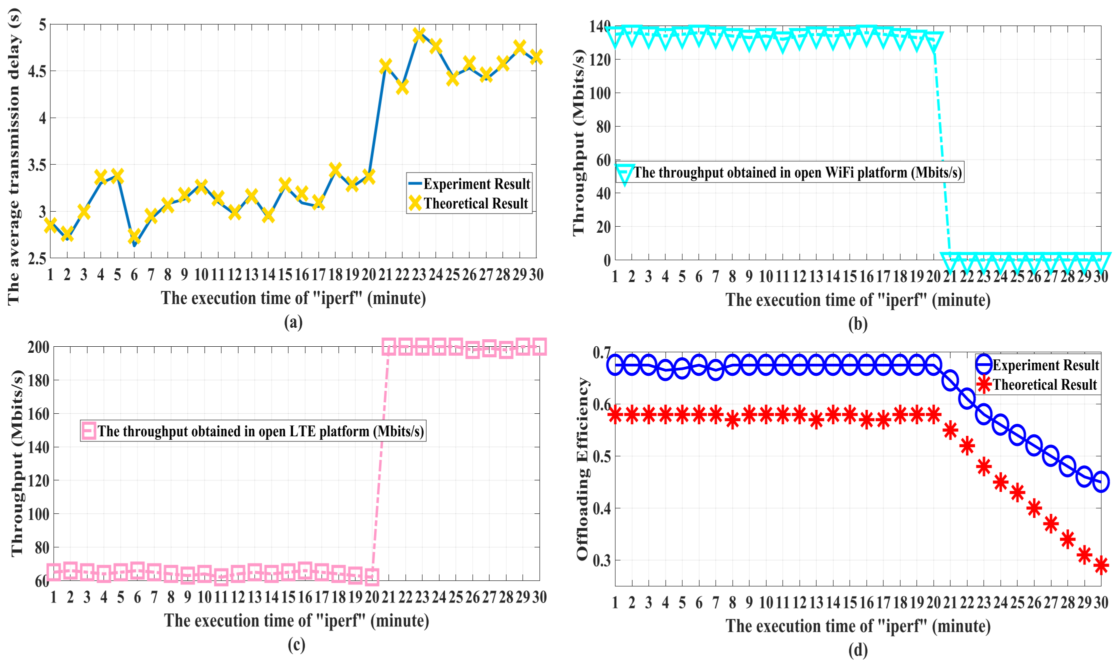


FIGURE 54. Experiment (solid lines) and theoretical (markers) results when the packets arrival rate in UE side is 200Mbits/s (i.e., "iperf" = 200Mbits/s), the execution time of "iperf" = 30 minute, the system utilization of open WiFi platform and open LTE platform = 15%, and the off period of open WiFi platform = 10 minute. (a) The average transmission delay; (b) The throughput obtained in the open WiFi platform; (c) The throughput obtained in the open LTE platform; (d) The offloading efficiency.

parameter setting (i.e., the packets arrival rate in UE side is set to 200Mbits/s, the execution time of "iperf" = 30 minute, the system utilization of open WiFi platform = 15%, and the system utilization of open LTE platform = 15%), and under the condition that the off period of open WiFi platform change from 5 minutes to 15 minutes. Compared with

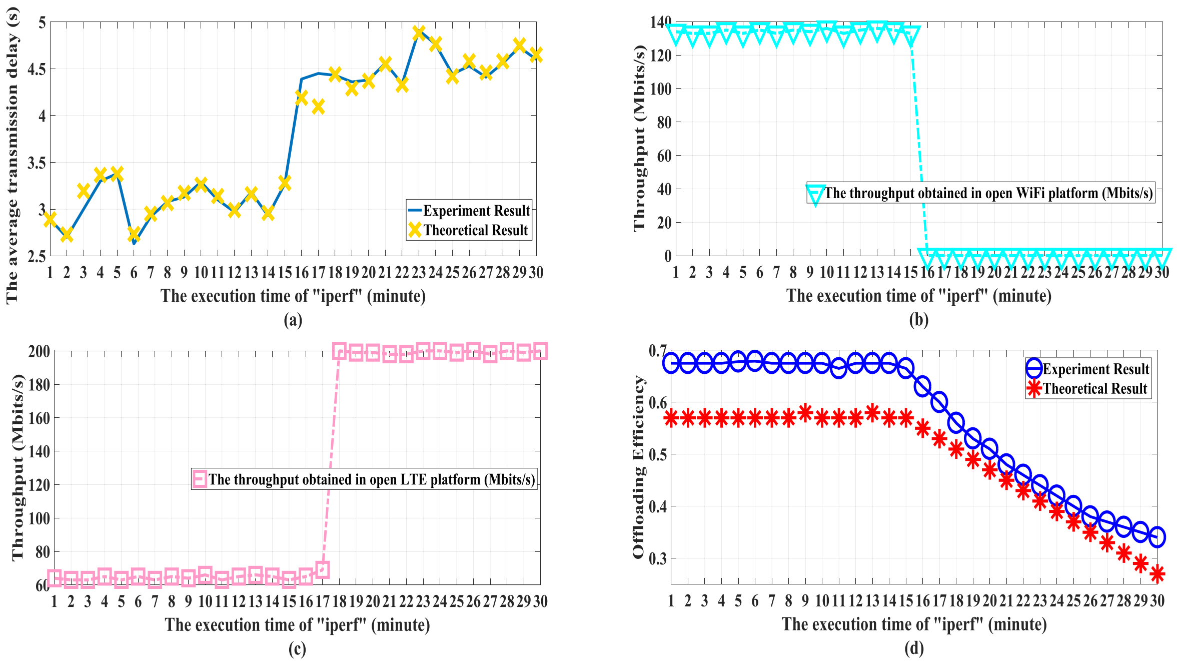


FIGURE 55. Experiment (solid lines) and theoretical (markers) results when the packets arrival rate in UE side is 200Mbits/s (i.e., "iperf" = 200Mbits/s), the execution time of "iperf" = 30 minute, the system utilization of open WiFi platform and open LTE platform = 15%, and the off period of open WiFi platform = 15 minute. (a) The average transmission delay; (b) The throughput obtained in the open WiFi platform; (c) The throughput obtained in the open LTE platform; (d) The offloading efficiency.

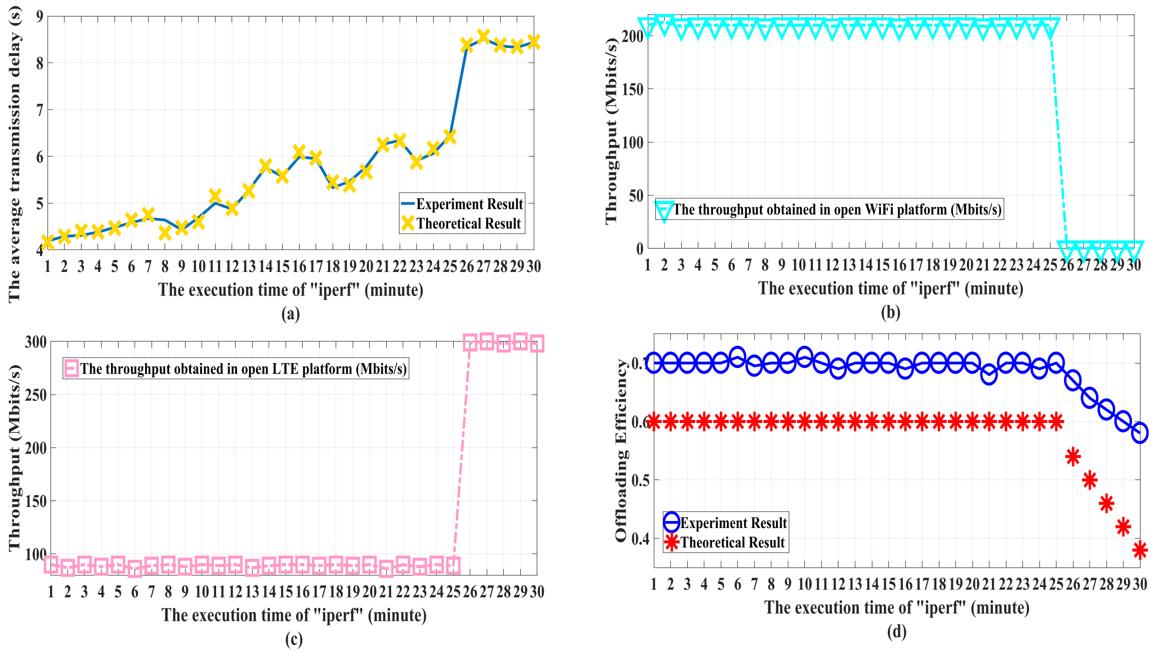


FIGURE 56. Experiment (solid lines) and theoretical (markers) results when the packets arrival rate in UE side is 300Mbits/s (i.e., "iperf" = 300Mbits/s), the execution time of "iperf" = 30 minute, the system utilization of open WiFi platform and open LTE platform = 20%, and the off period of open WiFi platform = 5 minute. (a) The average transmission delay; (b) The throughput obtained in the open WiFi platform; (c) The throughput obtained in the open LTE platform; (d) The offloading efficiency.

Fig.50 (a)-Fig.52 (a), we see that the average transmission delay of both the single path and the dual path follow ((Fig.53(a)-Fig.55(a)) > (Fig.50(a)-Fig.52(a)), this is because with the packets arrival rate in the UE side increase (i.e., increase to 200 Mbits/s), the average transmission delay increases accordingly. Furthermore, the theoretical transmission delay is roughly aligned with experiment results in all considered cases. The reason for some deviations between the

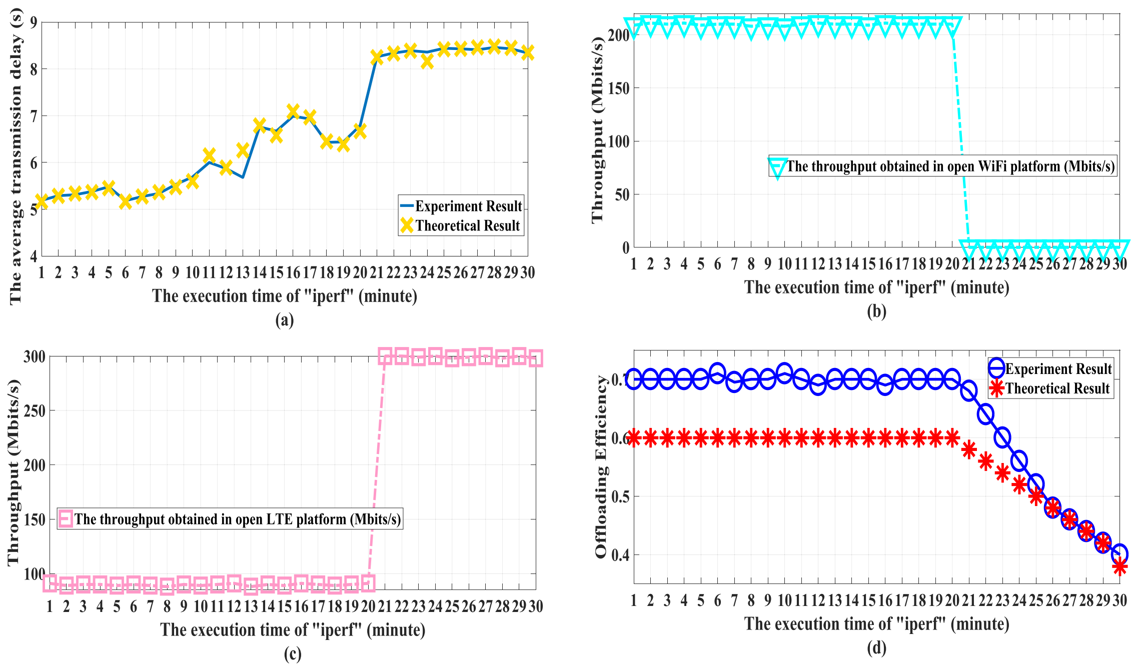


FIGURE 57. Experiment (solid lines) and theoretical (markers) results when the packets arrival rate in UE side is 300Mbits/s (i.e., "iperf" = 300Mbits/s), the execution time of "iperf" = 30 minute, the system utilization of open WiFi platform and open LTE platform = 20%, and the off period of open WiFi platform = 10 minute. (a) The average transmission delay; (b) The throughput obtained in the open WiFi platform; (c) The throughput obtained in the open LTE platform; (d) The offloading efficiency.

theoretical and experiment results is same as that illustrated in previous section.

2) THROUGHPUT OBTAINED IN OPEN WIFI PLATFORM

Fig.53 (b), Fig.54 (b), and Fig.55 (b) depict the throughput obtained in the open WiFi platform under the second group experiment parameter setting, and under the condition that the off period of open WiFi platform change from 5 minutes to 15 minutes. According to these three figures, we can note that when in the dual path offloading state, the actual throughput obtained in the open WiFi platform is approximately equal to 140 Mbits/s and therefore the actual throughput obtained in the open LTE platform is approximately equal to 60 Mbits/s. Compared with the results shown in Fig.50(b)-Fig.52(b) (i.e., 80Mbits/s obtained in WiFi and 20Mbites/s obtained in LTE), the actual throughput ratio is decrease (i.e., $140/60 < 80/20$). The reason contributes to it is that the system utilization of open WiFi platform and open LTE platform has increased from 10% to 15%. Beside the dual path offloading state, when in the single one path state, the actual throughput in the WiFi platform is equal to 0Mbits/s and the actual throughput in the LTE platform almost equals to 200Mbits/s.

3) THROUGHPUT OBTAINED IN OPEN LTE PLATFORM

Fig.53 (c), Fig.54 (c), and Fig.55 (c) illustrate the throughput obtained in the open LTE platform under the second group experiment parameter setting, and under the condition that the off period of open WiFi platform change from 5 minutes to 15 minutes. From the three figures, we can see when in the dual path offloading state, the actual throughput obtained in the open LTE platform is approximately equal to 60 Mbits/s, this experiment results are corresponding to the results shown in Fig.53(b)-Fig.55(b) (i.e., the total is 200 Mbits/s). Besides, as mentioned above, when in the single one path state, the actual throughput in the open LTE platform almost equals to 200Mbits/s.

4) OFFLOADING EFFICIENCY

Fig.53 (d), Fig.54 (d), and Fig.55 (d) demonstrate the offloading efficiency under the second group experiment parameter setting, and under the condition that the off period of open WiFi platform change from 5 minutes to 15 minutes. Based on the three figures, we can find the offloading efficiency of both the single path and the dual path follow (Fig.53 (d)-Fig.55 (d)) $<$ (Fig.50 (d)-Fig.52 (d)). This is because with the system utilization increase, the offloading efficiency decreases accordingly. Moreover, the theoretical transmission delay is roughly aligned with experiment results in all considered cases. The reason for some deviations between the theoretical and experiment results is same as that mentioned in previous section.

C. PERFORMANCE UNDER THE THIRD GROUP EXPERIMENT PARAMETER SETTING

In this sub-section, we carry out the third group experiment to evaluate the analytical results proposed in section IV. The experiment procedures are the same as that described

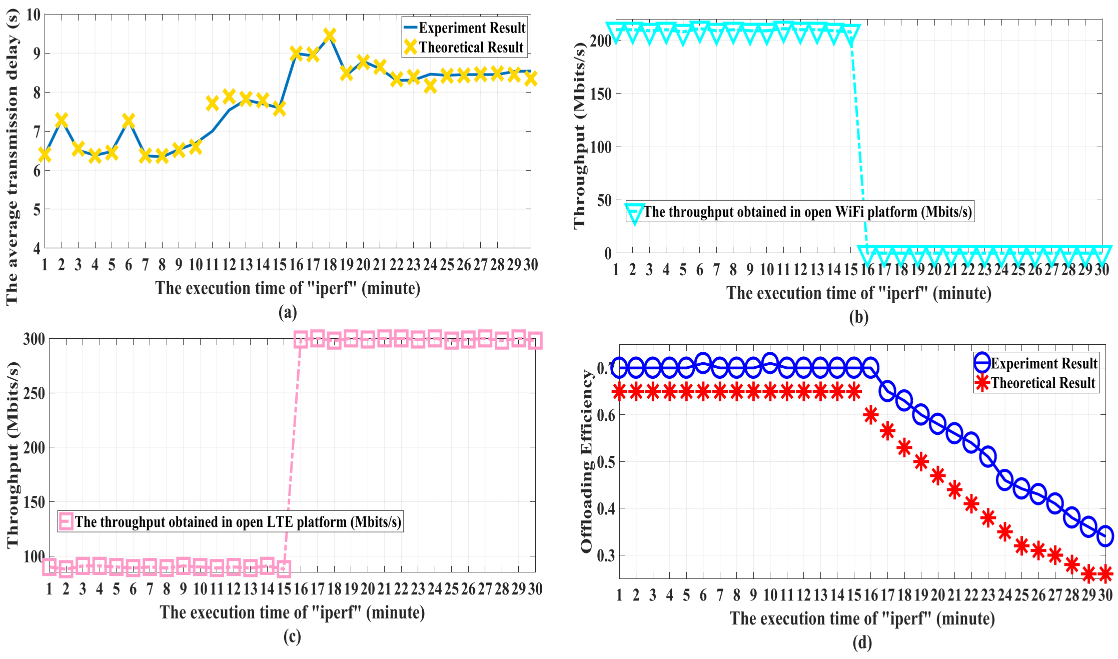


FIGURE 58. Experiment (solid lines) and theoretical (markers) results when the packets arrival rate in UE side is 300Mbits/s (i.e., "iperf" = 300Mbits/s), the execution time of "iperf" = 30 minute, the system utilization of open WiFi platform and open LTE platform = 20%, and the off period of open WiFi platform = 15 minute. (a) The average transmission delay; (b) The throughput obtained in the open WiFi platform; (c) The throughput obtained in the open LTE platform; (d) The offloading efficiency.

in section V.A and section V.B. Based on the third group experiment results, we draw the following three statistic figures which are shown from Fig.56-Fig.58.

1) AVERAGE TRANSMISSION DELAY

Fig.56 (a), Fig.57 (a), and Fig.58 (a) depict the average transmission delay under the third group experiment parameter setting (i.e., the packets arrival rate in UE side is set to 300Mbits/s, the execution time of "iperf" = 30 minute, the system utilization of open WiFi platform = 20%, and the system utilization of open LTE platform = 20%), and under the condition that the off period of open WiFi platform change from 5 minutes to 15 minutes. Compared with Fig.50 (a)-Fig.52 (a) and Fig.53 (a)-Fig.55 (a), we observe that the average transmission delay of both the single path and the dual path follow ((Fig.56 (a)-Fig.58(a)) > (Fig.53 (a)-Fig.55 (a)) > (Fig.50 (a)-Fig.52 (a)), this is because with the packets arrival rate in the UE side increase (i.e., increase to 300 Mbits/s), the average transmission delay increase accordingly. Furthermore, the theoretical transmission delay is roughly aligned with experiment results in all considered cases. The reason for some deviations between the theoretical and experiment results is same as that described in previous section.

2) THROUGHPUT OBTAINED IN OPEN WIFI PLATFORM

Fig.56(b), Fig.57 (b), and Fig.58 (b) demonstrate the throughput obtained in the open WiFi platform under the third group experiment parameter setting, and under the condition that the off period of open WiFi platform change from 5 minutes to 15 minutes. According to these three figures, we can see that when in the dual path offloading state, the actual throughput obtained in the open WiFi platform is approximately equal to 200 Mbits/s and therefore the actual throughput obtained in the open LTE platform is approximately equal to 100 Mbits/s. Compared with the results shown in Fig.53(b)-Fig.55(b) (i.e., 160Mbits/s obtained in WiFi and 40Mbites/s obtained in LTE) and in Fig.50 (b)-Fig.52(b) (i.e., 80Mbits/s obtained in WiFi and 20Mbites/s obtained in LTE), the actual throughput ratio is decrease (i.e., 200/100 < 140/60 < 80/20). The reason contributes to this phenomenon is that the system utilization of open WiFi platform and open LTE platform has increased from 10% to 20%. Beside the dual path offloading state, when in the single one path state, the actual throughput in the WiFi platform is equal to 0Mbits/s and the actual throughput in the LTE platform almost equals to 300Mbits/s.

3) THROUGHPUT OBTAINED IN OPEN LTE PLATFORM

Fig.56 (c), Fig.57 (c), and Fig.58 (c) show the throughput obtained in the open LTE platform under the third group experiment parameter setting, and under the condition that the off period of open WiFi platform change from 5 minutes to 15 minutes. Based on the three figures, we can see when in the dual path offloading state, the actual throughput obtained in the open LTE platform is approximately equal to 100 Mbits/s, this experiment results are corresponding to the results shown in Fig.56(b)-Fig.58(b) (i.e., the total is 300 Mbits/s). Besides, as mentioned above, when in the single one path state, the actual throughput in the open LTE platform almost equals to 300Mbits/s.

4) OFFLOADING EFFICIENCY

Fig.56 (d), Fig.57 (d), and Fig.58 (d) demonstrate the offloading efficiency under the third group experiment parameter setting, and under the condition that the off period of open WiFi platform change from 5 minutes to 15 minutes. Based on the three figures, we can find the offloading efficiency of both the single path and the dual path follow (Fig.53 (d)-Fig.55 (d)) < (Fig.56 (d)-Fig.58 (d)) < (Fig.50 (d)-Fig.52 (d)), which is an unexpected result. Strictly speaking, the offloading efficiency should decrease with the system utilization increase (i.e., the trend should be (Fig.56 (d)-Fig.58 (d)) < (Fig.53 (d)-Fig.55 (d)) < (Fig.50 (d)-Fig.52 (d))), the offloading efficiency decreases accordingly. One possible reason to explain this result is that maybe both the open WiFi platform and the open LTE platform have reached their uplink throughput capacity limit and therefor causing the unexpected offloading efficiency degradation. Besides, we can see the theoretical transmission delay is roughly aligned with experiment results in all considered cases. The reason for some deviations between the theoretical and experiment results is same as that mentioned in previous section.

VI. CONCLUSION

In order to depict the packets offloading scheme in LTE/WiFi HetNets scenario, in this paper, we analyze the key performance metrics for single path and dual path packets offloading with closed-form expressions based on software-defined open HetNets platform. To be specific, we first construct the software defined open HetNets platform. Based on software-defined open HetNets platform, we then propose an analytical model which has been derived for evaluating the packets offloading performance in terms of the packets transmission delay and offloading efficiency as functions of relevant system parameters. These expressions show that the packet arrival rate in UE and the WiFi connection duration have great influences on the packets offloading performance. Experiment results verify the accuracy of the proposed analytical model and closed-form theoretical analyses, which provide guidance for maximizing the network resource utilization efficiency as well as optimizing packets offloading performance in densely deployed LTE/WiFi HetNets.

Currently, the current version software-defined open WiFi platform and open LTE platform could not perfectly assist the enhanced EPC due to the MAC functions and PHY functions in those two platforms are still need to be further improved. Therefore, in the future, we will continue evolve the MAC and PHY functions in our novel open WiFi platform and open LTE platform, which we hope to further improve the HetNets platform's performance.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their helpful comments and suggestions for our work.

APPENDIX A
DERIVATION FROM EQUATION (5) AND EQUATION (6) TO EQUATION (7)

The derivation process from the second sub-equation in (5) and the second sub-equation in (6) to equation (7) is, (29), as shown at the bottom of this page.

APPENDIX B
DERIVATION FROM THE SECOND ITEM IN EQUATION (13) TO EQUATION (15)

Based on equation (14), the detailed derivation process from equation (13) to equation (15) is (31), as shown at the top of the next page.

$$\begin{aligned}
&\xi_{AVE}\\
&= (SP_{\cdot,CE} - SP_{0,CE})\theta_{CE} + (SP_{\cdot,WI} - SP_{0,WI})\theta_{WI}\\
&\Rightarrow \xi_{AVE} = SP_{\cdot,CE}\theta_{CE} - SP_{0,CE}\theta_{CE} + SP_{\cdot,WI}\theta_{WI} - SP_{0,WI}\theta_{WI}\\
&\Rightarrow \xi_{AVE} = SP_{\cdot,CE}\theta_{CE} + SP_{\cdot,WI}\theta_{WI} - SP_{0,CE}\theta_{CE} - SP_{0,WI}\theta_{WI}\\
&\Rightarrow \xi_{AVE} = \theta_{AVE} - SP_{0,CE}\theta_{CE} - SP_{0,WI}\theta_{WI}\\
&\Rightarrow SP_{0,CE}\theta_{CE} + SP_{0,WI}\theta_{WI} = \theta_{AVE} - \xi_{AVE}
\end{aligned} \tag{30}$$

$$\begin{aligned}
&\begin{cases} SP_{0,CE}Z^0(\xi_C + \beta_C + \theta_{CE}) = SP_{0,WI}\beta_W Z^0 + SP_{1,CE}\theta_{CE}Z^0 + SP_{0,CE}\theta_{CE}Z^0, & (j = 0)\\ SP_{j,CE}Z^j(\xi_C + \beta_C + \theta_{CE}) = SP_{j-1,CE}\xi_C Z^j + SP_{j,WI}\beta_W Z^j + SP_{j+1,CE}\theta_{CE}Z^j, & (j > 0)\end{cases}\\
&\text{left} = (\xi_C + \beta_C + \theta_{CE})(SP_{0,CE}Z^0 + SP_{1,CE}Z^1 + \cdots + SP_{j,CE}Z^j + \cdots SP_{n,CE}Z^n)(n = \infty)\\
&\quad = (\xi_C + \beta_C + \theta_{CE})G_{CE}(Z)\\
&\text{right} = (\sum_{j=0}^{\infty} SP_{j,WI}\beta_W Z^j) + (\sum_{j=1}^{\infty} SP_{j-1,CE}\xi_C Z^j) + (SP_{0,CE}\theta_{CE}Z^0 + SP_{1,CE}\theta_{CE}Z^0 + \sum_{j=1}^{\infty} SP_{j+1,CE}\theta_{CE}Z^j)\\
&\quad = (\beta_W G_{WI}(Z)) + ((SP_{1-1,CE}\xi_C Z^0)Z + (SP_{2-1,CE}\xi_C Z^1)Z + \cdots (SP_{n-1,CE}\xi_C Z^{n-1})Z)\\
&\qquad + (SP_{0,CE}\theta_{CE}Z^0 + \frac{SP_{1,CE}\theta_{CE}Z^0 Z + SP_{1+1,CE}\theta_{CE}Z^1 Z + \cdots SP_{n+1,CE}\theta_{CE}Z^n}{Z})\\
&\quad = (\beta_W G_{WI}(Z)) + (\xi_C Z G_{CE}(Z)) + (SP_{0,CE}\theta_{CE} + \frac{\theta_{CE}}{Z}(G_{CE}(Z) - SP_{0,CE}))\\
&\quad = (SE_2) + (SE_1) + (SE_4 + SE_3)\\
&\text{left} = \text{right}
\end{aligned} \tag{29}$$

$$
\begin{aligned}
&\frac{\theta_{AVE} - \xi_{AVE} - SP_{0,CE}\theta_{CE}}{\theta_{WI}}\beta_W\theta_{WI}Z_0 + SP_{0,CE}\theta_{CE}[\beta_W Z_0 + \xi_W Z_0(1-Z_0) - \theta_{WI}(1-Z_0)] = 0\\
&\Rightarrow (\theta_{AVE} - \xi_{AVE} - SP_{0,CE}\theta_{CE})\beta_W Z_0 + SP_{0,CE}\theta_{CE}[\beta_W Z_0 + \xi_W Z_0(1-Z_0) - \theta_{WI}(1-Z_0)] = 0\\
&\Rightarrow \theta_{AVE}\beta_W Z_0 - \xi_{AVE}\beta_W Z_0 - SP_{0,CE}\theta_{CE}\beta_W Z_0 + SP_{0,CE}\theta_{CE}[\beta_W Z_0 + \xi_W Z_0 - \xi_W Z_0^2 - \theta_{WI} + \theta_{WI}Z_0] = 0\\
&\Rightarrow SP_{0,CE}\theta_{CE}[\beta_W Z_0 + \xi_W Z_0 - \xi_W Z_0^2 - \theta_{WI} + \theta_{WI}Z_0 - \beta_W Z_0] + \theta_{AVE}\beta_W Z_0 - \xi_{AVE}\beta_W Z_0 = 0\\
&\Rightarrow SP_{0,CE}\theta_{CE}[\beta_W Z_0 + \xi_W Z_0 - \xi_W Z_0^2 - \theta_{WI} + \theta_{WI}Z_0 - \beta_W Z_0] = \xi_{AVE}\beta_W Z_0 - \theta_{AVE}\beta_W Z_0\\
&\Rightarrow SP_{0,CE} = \frac{\beta_W Z_0(\xi_{AVE} - \theta_{AVE})}{[\xi_W Z_0 - \xi_W Z_0^2 - \theta_{WI} + \theta_{WI}Z_0]\theta_{CE}} = \frac{\beta_W Z_0(\xi_{AVE} - \theta_{AVE})}{[(1-Z_0)(\xi_W Z_0 - \theta_{WI})]\theta_{CE}}\\
&\Rightarrow SP_{0,CE} = \frac{\beta_W Z_0(\theta_{AVE} - \xi_{AVE})}{[(1-Z_0)(\theta_{WI} - \xi_W Z_0)]\theta_{CE}} \qquad (31)
\end{aligned}
$$

APPENDIX C
DERIVATION FROM EQUATION (11) AND EQUATION (15) TO EQUATION (16)

The detailed derivation process from equation (11) and equation (15) to equation (16) is shown below. We use equation (15) to replace the $SP_{0,WI}$ in equation (11), then we can obtain the solutions of $SP_{0,WI}$ and $SP_{0,CE}$.

Specifically, the calculation process for $SP_{0,CE}$ is shown below:

Then, using the same calculation, we can get the solution of $SP_{0,WI}$.

REFERENCES

[1] P.-Y. Kong and G. K. Karagiannidis, "Backhaul-aware joint traffic offloading and time fraction allocation for 5G HetNets," *IEEE Trans. Veh. Technol.*, vol. 65, no. 11, pp. 9224–9235, Nov. 2016.

[2] Q. Chen, G. Yu, H. Shan, A. Maaref, G. Y. Li, and A. Huang, "Cellular meets WiFi: Traffic offloading or resource sharing?" *IEEE Trans. Wireless Commun.*, vol. 15, no. 5, pp. 3354–3367, May 2016.

[3] D. Suh, H. Ko, and S. Pack, "Efficiency analysis of WiFi offloading techniques," *IEEE Trans. Veh. Technol.*, vol. 65, no. 5, pp. 3813–3817, May 2016.

[4] H. Ko, J. Lee, and S. Pack, "Performance optimization of delayed WiFi offloading in heterogeneous networks," *IEEE Trans. Veh. Technol.*, vol. 66, no. 10, pp. 9436–9447, Oct. 2017.

[5] S.-I. Sou and Y.-T. Peng, "Performance modeling for multipath mobile data offloading in cellular/Wi-Fi networks," *IEEE Trans. Commun.*, vol. 65, no. 9, pp. 3863–3875, Sep. 2017.

[6] J. Jia, G. Liu, D. Han, and J. Wang, "A novel packets transmission scheme based on software defined open wireless platform," *IEEE Access*, vol. 6, pp. 17093–17118, 2018.

[7] J. Jia, G. Liu, D. Han, and J. Wang, "Towards studying the two-tier intra-frequency X2 handover based on software-defined open LTE platform," *IEEE Access*, vol. 6, pp. 39643–39684, 2018.

[8] M. A. P. González, T. Higashino, and M. Okada, "Radio access considerations for data offloading with multipath TCP in cellular/WiFi networks," in *Proc. Int. Conf Inf. Netw. (ICOIN)*, Jan. 2013, pp. 28–30.

[9] F. Rebecchi, M. D. de Amorim, V. Conan, A. Passarella, R. Bruno, and M. Conti, "Data offloading techniques in cellular networks: A survey," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 2, pp. 580–603, 2nd Quart., 2015.

[10] S. Dimatteo, H. Pan, H. Bo, and V. K. O. Li, "Cellular traffic offloading through WiFi networks," in *Proc. IEEE 8th Int. Conf Mobile Adhoc Sensor Syst. (MASS)*, Oct. 2011, pp. 192–201.

[11] K. Lee, J. Lee, Y. Yi, I. Rhee, and S. Chong, "Mobile data offloading: How much can WiFi deliver?" *IEEE/ACM Trans. Netw.*, vol. 21, no. 2, pp. 536–550, Apr. 2013.

[12] M. Bennis, M. Simsek, A. Czylwik, W. Saad, S. Valentin, and M. Debbah, "When cellular meets WiFi in wireless small cell networks," *IEEE Commun. Mag.*, vol. 51, no. 6, pp. 44–50, Jun. 2013.

[13] L. Gao, G. Iosifidis, J. Huang, L. Tassiulas, and D. Li, "Bargaining-based mobile data offloading," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 6, pp. 1114–1125, Jun. 2014.

[14] B. Han, P. Hui, V. S. A. Kumar, M. V. Marathe, J. Shao, and A. Srinivasan, "Mobile data offloading through opportunistic communications and social participation," *IEEE Trans. Mobile Comput.*, vol. 11, no. 5, pp. 821–834, May 2012.

[15] A. Pyattaev, K. Johnsson, S. Andreev, and Y. Koucheryavy, "3GPP LTE traffic offloading onto WiFi direct," in *Proc. IEEE Wireless Commun. Netw. Conf. Workshops*, Apr. 2013, pp. 135–140.

[16] S. Singh, H. S. Dhillon, and J. G. Andrews, "Offloading in heterogeneous networks: Modeling, analysis, and design insights," *IEEE Trans. Wireless Commun.*, vol. 12, no. 5, pp. 2484–2497, May 2013.

[17] J. Lee, Y. Yi, S. Chong, and Y. Jin, "Economics of WiFi offloading: Trading delay for cellular capacity," in *Proc. IEEE Conf. Comput. Commun. Workshops*, Apr. 2013, pp. 357–362.

[18] D. Kim, Y. Noishiki, Y. Kitatsuji, and H. Yokota, "Efficient ANDSF-assisted Wi-Fi control for mobile data offloading," in *Proc. 9th Int. Wireless Commun. Mobile Comput. Conf.*, Jul. 2013, pp. 343–348.

[19] A. Y. Ding *et al.*, "Enabling energy-aware collaborative mobile data offloading for smartphones," in *Proc. IEEE Int. Conf. Sens., Commun. Netw.*, Jun. 2013, pp. 487–495.

[20] J. Kim, N.-O. Song, B. H. Jung, H. Leem, and D. K. Sung, "Placement of WiFi access points for efficient WiFi offloading in an overlay network," in *Proc. IEEE 24th Annu. Int. Symp. Pers. Indoor Mobile Radio Commun.*, Sep. 2013, pp. 3066–3070.

[21] J. R. Iyengar, P. D. Amer, and R. Stewart, "Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths," *IEEE/ACM Trans. Netw.*, vol. 14, no. 5, pp. 951–964, Oct. 2006.

[22] S. J. Koh, M. J. Chang, and M. Lee, "MSCTP for soft handover in transport layer," *IEEE Commun. Lett.*, vol. 8, no. 3, pp. 189–191, Mar. 2004.

[23] H.-Y. Hsieh and R. Sivakumar, "A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts," *Wireless Netw.*, vol. 11, nos. 1–2, pp. 99–114, Jan. 2005.

[24] T. You, C. Park, H. Jung, T. T. Kwon, and Y. Choi, "Multipath transmission architecture for heterogeneous wireless networks," in *Proc. ICTC*, 2011, pp. 26–31.

[25] C. Paasch, G. Detal, F. Duchene, C. Raiciu, and O. Bonaventure, "Exploring mobile/WiFi handover with multipath TCP," in *Proc. ACM SIGCOMM Workshop Cellular Netw., Oper., Challenges, Future Design*, 2012, pp. 31–36.

[26] C. Raiciu, D. Niculescu, M. Bagnulo, and M. J. Handley, "Opportunistic mobility with multipath TCP," in *Proc. 6th Int. Workshop MobiArch*, 2011, pp. 7–12.

[27] U. Yechiali and P. Naor, "Queuing problems with heterogeneous arrivals and service," *Oper. Res.*, vol. 19, no. 3, pp. 722–734, 1971.

[28] S. M. Ross, *Stochastic Processes*, 2nd ed. Hoboken, NJ, USA: Wiley, 1996.

GUANGZHONG LIU received the B.S. degree from Southwest Jiaotong University and the Ph.D. degree from the China University of Mining and Technology. He is currently a Professor of computer science and engineering with Shanghai Maritime University. His specific research interests include underwater acoustic communication technology, mobile networking, wireless communication, and network security.

JIANXIN JIA received the B.S. degree in computer science and technology from the Qingdao University of Technology and the M.S. degree in computer science and engineering from Shanghai Maritime University, where he is currently pursuing the Ph.D. degree. In 2013, he received the Qualification Certificate of Computer and Software Technology Proficiency as a Database System Engineer, the Qualification Certificate of Computer and Software Technology Proficiency as a Networking Engineer, the Qualification Certificate of Computer and Software Technology Proficiency as a Software Designer in 2014, the Qualification Certificate of Computer and Software Technology Proficiency as a Software Testing Engineer in 2015, and the Honorable Certificate related to the Qualification Certificate of Computer and Software Technology Proficiency (top 3 in Zhejiang province) in 2016. His main research interests include wireless communication, distributed computing, mobile networking, underwater acoustic communication technology, and cloud security.