Low Latency Queuing Control in Extendable Mobile Ad-hoc Network Emulator (EMANE)

by

Shaoyi Li

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
December 14, 2019

Keywords: Scheduling in packet level, Multi-objective latency minimization, single-hop and multi-hops network

Abstract

A variation packets delay which is also called packet jitter causes packet loss and delay. When jitter happens to stream video, users suffer an annoying freezes which results in user unpleasant. The solution for reducing jitter probability is a low latency low-complexity scheduling policy in queuing system: the Earliest Due Date first (EDD) policy.

To evaluate the performance of EDD in a real-world wireless network system. We implement EDD algorithms on Extendable Mobile Ad-hoc Network Emulator (EMANE) to take virtual field tests. In this paper, we present the design and implementation of a low-complexity queue scheduling module in both single-hop multi-server systems using C/C++. We also show an efficient debug method in EMANE development. Our emulation results demonstrate the EDD policy achieves over 1000x reduction in jitter probability compared to the commonly used FCFS policy.

Acknowledgments

Firstly, I would like to express my great appreciation to my advisor Prof. Yin Sun during my master degree. Prof. Yin Sun encouraged and helped me a lot to overcome a large number of difficulties I can't face myself. Even with a poor knowledge foundation at beginning of my research, Prof. Yin Sun gave me guidance patiently and trained me carefully which let me make much progress. I always feel lucky to be his students. In addition, I would like to thank my committee members, Prof. Shiwen Mao, and Prof Xiaowen Gong. They gave me lots of suggestions and helpful comment, so that I can modify and improve my thesis research.

Besides my committee members, I am thankful for my lab mate Kamran Chowdhury Shisher who provides me helps in my oral presentation. I learn a lot from them, I will remember the happy time we work in the lab together.

Last but not the least, I want to express my thanks to my parents, who raised me up and gave me tremendous support for my studying in United States. It is their strong support that helps me complete my Master study.

Table of Contents

List of Figures

Chapter 1

Introduction

Network latency is how much time it takes for a network to receive a control message and the transmission actually occurs. Low network latency is preferred in our daily life.

Previous studies demonstrated that the changes in packet transmission order by different policies reduced different kinds of latency [1], [2]. Earliest Due Date (EDD) First policy minimizes jitter probability, First-Come, First-Served (FCFS) policy minimizes maximum delay, respectively [1], [2]. Of these scheduling policies, FCFS is the most common policy used in transmitter MAC layer queuing buffer models; however this policy is not delay-optimal for all delay metrics in a real network system. Therefore, it is necessary to evaluate the performance of different polices in queuing delay minimization using simulation and emulation tools.

There are two different approaches of network performance evaluations: hardware emulation [3] and software solutions.[4], [5], [6]. [7]. The software simulation and emulation environment is easier to control and the expense is significantly less when compared with the hardware testbed. The most common network software simulators and emulators are NS2/NS3 [8], Common Open Research Emulator (CORE) [9], and Extendable Mobile Ad-hoc Network Emulator (EMANE) [5]. With these, NS2/NS3 is a discrete-event network simulator for Internet systems, targeted primarily for research and educational use [8]. “However, NS2/NS3 simulators are incapable of providing statistics of real-time performance, these applications change the time base (for example, 1 ms of simulation is done in a much longer duration) and are not real time, whereas network “emulation” works in real time [10]. ” Thus, NS2/NS3 is not suitable for our real-time latency optimal queuing system. In contrast, CORE is a real-time emulator focusing on the upper three layers in Open System Interconnection Model (OSI) model. The emulator provides a convenient GUI for users to configure the virtual network,

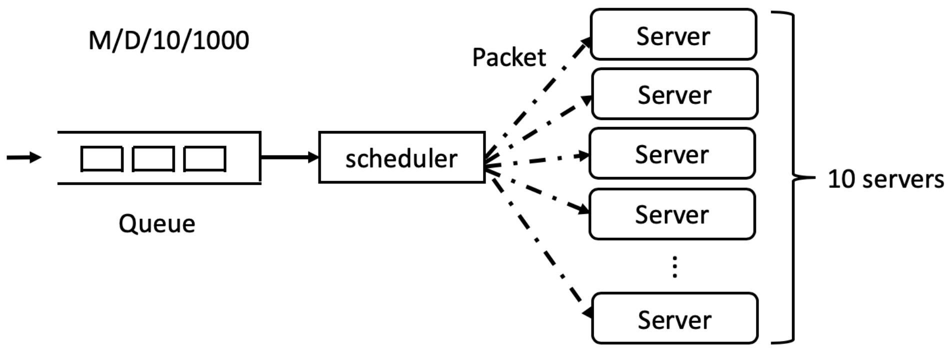


Figure 1.1: EMANE Queuing System

but the network scheduling happens in MAC layer, CORE doesn't contain this layer. With the goal of seeking real-time latency optimal queuing system, we choose EMANE to evaluate our scheduling policies performance since EMANE mainly emulates the MAC and PHY layer in OSI model and this emulator provides real-time statistic which meets our study perfectly.

1.1 Design and Implementation of low latency Queuing system

In this thesis, we studied a low-complexity latency-optimal queuing control scheduling module in Extendable Mobile Ad-hoc Network Emulator (EMANE) with TDMA channel access module. The queuing system is a centralized multiple servers single-hop system which used D/M/10/1000 model as the queue model shown in Fig 1.1. The arrival process is a Markovian - Poisson process with a rate λ = 66 packets per second, the service distribution is deterministic which has a fixed inter-arrivals with a rate μ = 70 packets per second to control system to be stable, we used a 1000 size queuing buffer and 10 single servers. We compared two different scheduling policies EDD and FCFS in jitter probability minimization.

Since EMANE has long compilation time which is usually longer than 15 minutes and it lacks of debug mode which is highly difficult for packets and queue structure parsing, we firstly migrated the original EMANE queue structure into a transparent Integrated Development Environment (IDE) Xcode, which exceedingly reduces the workload and difficulties in compiling and debugging.

We then build the low complexity EDD queue module in Xcode using C++ standard data structure list whose size is 1000, the algorithm complexity of insertion sort $O(n)$ is much less than the complexity of bubble sort $O(n^2)$ using array [11]. We test the EDD queue module in Xcode with Poisson arrival process and exponential service times.

Later, we move the EDD queue module into EMANE transmitter MAC layer, and we assign a absolute due time for every packet in its virtual header when the packet comes to the queue buffer in EMANE, the packet order can be changed in the EDD queue module according to the due time when they come into the queue, and earliest due time packet move out of the queue firstly.

We next configure the arrival process by a network traffic flow generator, Multi-Generator (MGEN) [12] using specific command line, we use EMANE's original deterministic service distribution and keep the queue system stable, 10000 packets are generated into the traffic flow.

Finally, we collect packets' completion time at transmitter when the packets finish scheduling in EMANE's data log file *emane.log*, we use a Python keywords crawler getting due time and completion time, then we simulate the jitter probability of EMANE's original FCFS queuing policy and our EDD policy based on these data with MATLAB. Our emulation results demonstrate EDD policy can achieve over 1000x reduction in jitter probability compared to the common FCFS policy. Please see `https://github.com/szl0144/Multi-hop` for queuing control source code in EMANE.

1.2 Layout

We describe the research background in the Chapter 2, which introduces MGEN traffic flows generator and EMANE framework. We introduce our low latency low-complexity single-hop Queuing control system in Chapter 3. Performance evaluation is presented in Chapter 4. We provide a future design for latency optimal multi-hop Queuing control system in Chapter 5. Finally, Chapter 6 reviews conclusion.

Chapter 2

Introduction of MGEN and EMANE

2.1 Introduction of MGEN Version 5.02

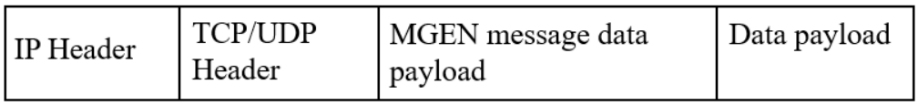
The Multi-Generator (MGEN) version 5.02 is a open source software generating UDP or TCP IP network bytes traffic flow [13], MGEN creates IP datagram for the down layer emulator to perform network test, such as NS2/NS3 and EMANE. MGEN can be controlled by a script file, the operator write input into MGEN script file by different command lines to take traffic flow control operations.

MGEN can generate the traffic flow in a downlink way and receive them in a uplink way, all the information of traffic flow is record in mgen.log file, this file helps us trouble shooting the bugs when we configure the traffic flow and know the details in packet. It's also record the generation time and arrival time of each datagram, this helps us calculate the latency of the network system [14].

2.1.1 Format Of the Traffic Flow Datagram

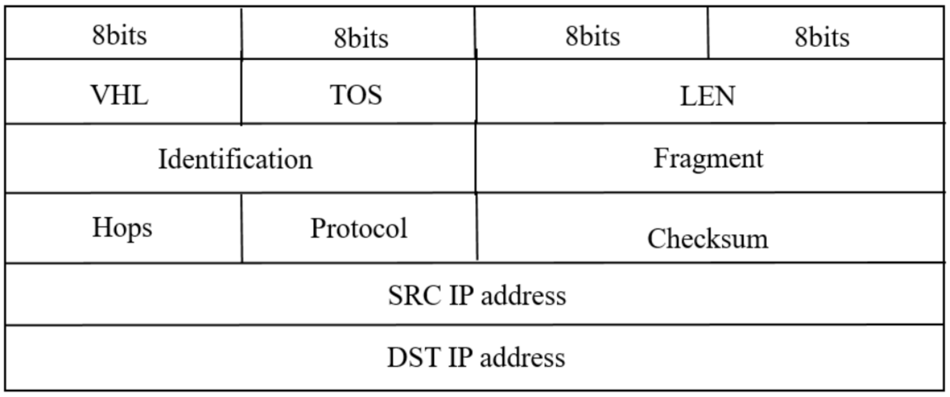
The format of traffic flow datagram is important for the users who focuses on the data in the packet, the traffic flow generated from MGEN is thousands of binary bits, it is impossible to locate the part you want in packet if you don't know the format of the output packet, it encourages us to study the format of the datagram.

The format of traffic datagram is shown in Fig 2.1. The output from MGEN include the IP header, TCP/UDP header, MGEN message data payload and Data payload information, that means MGEN emulates the upper three layers, Application layer, Transport layer and Network layer in OSI model [15].



IP Header	TCP/UDP Header	MGEN message data payload	Data payload

Figure 2.1: MGEN Output Datagram Format Generated from [13]



8bits	8bits	8bits	8bits
VHL	TOS	LEN	
Identification		Fragment	
Hops	Protocol	Checksum	
SRC IP address			
DST IP address			

Figure 2.2: IPv4 Header in MGEN Datagram Generated from [16]

MGEN creates its own special datagram, the structure details are introduced in the following part, because we use IPv4 and UDP protocol in our project, we introduce these two header information in this thesis.

IPv4 Header format in MGEN

The first part of MGEN datagram is IPv4 Header, the IPv4 Header structure is shown in Fig 2.2. The data bits of IP header from MGEN output matches the real IPv4 header [17], it contains 10 fields. Deserve to be mentioned, all bytes transmitted in the traffic flow are in network bytes order also called big endian, "the most significant value in the sequence is stored at the lowest storage address [18]." To transfer network bytes order to host bytes order, we need to use C++ function $ntohs()$ [19].

The first 8 bits is version and header length (VHL) part, VHL contains the 4 bits internet protocol version and 4 bits internet header length information, we set the version as 0100 which represent IPv4 protocol, the internet header length is 1001 which means the header length is 20 bytes. The second 8 bits field in the IPv4 header is type of service (TOS), we just use the first 6 bits in this part, which shows the datagram priority in the network, we set TOS part as all 0s, which means normal service. The third field is datagram length, the length consists of IPv4

16bits	16bits
DST	SRC
LEN	Check

Figure 2.3: UDP Header in MGEN Datagram Generated from [22]

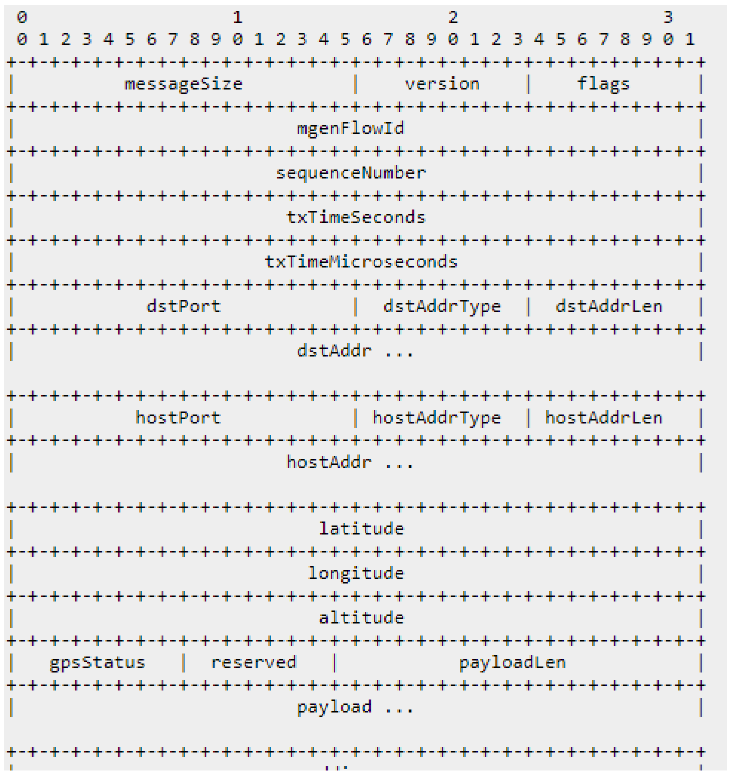
header, UDP header and datagram payload length, the length is larger than 20 bytes and should be less than 2^{16} = 65536 bytes.The forth field is packet identification (ID), the ID is uniquely assigned to each packet, this field is used to identifying packet and IP fragments [20]. The fifth field is used for IP fragmentation and reassembling, we don't use this field in our project. The sixth field Hops means Time To Live (TTL), this filed counts how many hop does the packet go across, the value increases 1 when the packet go across a node. The seventh field Protocol shows what kind of protocol does the payload of the IPv4 datagram use, because we use UDP in the transport layer, the value is 17 according to the list of IP protocol numbers [21]. The eighth field checksum is used for error check in router. The last two fields are source IPv4 address we set in the MGEN script, and destination IPv4 address we set in script corresponding to the address of transmitter in emulator. The two addresses are not changed by network address translation device after we check this field in MGEN output data stream.

UDP Header format in MGEN

The second part in the MGEN datagram is UDP header shown in Fig 2.3, the first 16 bits field is source port number and the second 16 bits field is destination port number, each port number has its own description, the well-known port number is described in the list [23]. We set both of the source port and destination port to be 5001 in MGEN. The third field is the length of the UDP segment, and the last field is the checksum field for data error-checking.

MGEN message data payload

The payload of datagram of MGEN called MGEN message data payload [13], the format of this part is shown in Fig 2.4, this part is used to message recognized at MGEN receiver, the receiver at application layer read the bits data from the beginning of the datagram payload part



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          messageSize          |    version    |     flags     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          mgenFlowId                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        sequenceNumber                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         txTimeSeconds                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       txTimeMicroseconds                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            dstPort            |  dstAddrType  |  dstAddrLen   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          dstAddr ...                          |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           hostPort            | hostAddrType  | hostAddrLen   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         hostAddr ...                          |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            latitude                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           longitude                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            altitude                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   gpsStatus   |   reserved    |          payloadLen           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          payload ...                          |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

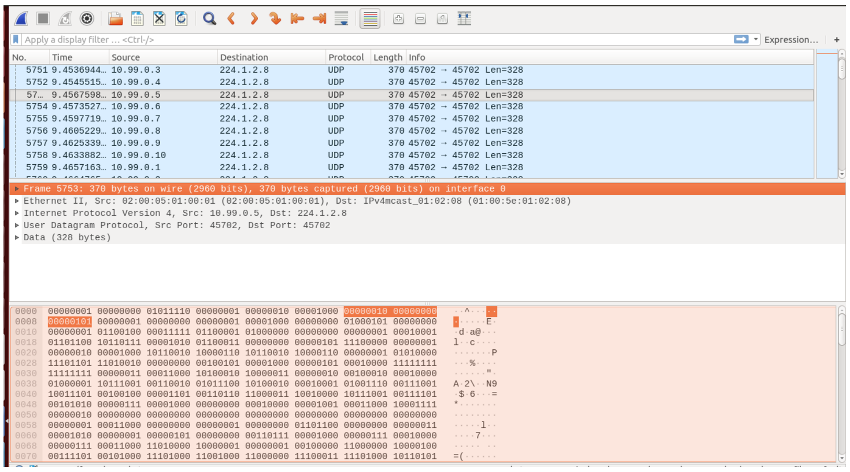
Figure 2.4: Message Data Payload in MGEN Datagram. Source [13]

and extract the information from each field in order, the data payload of datagram is contained in the message payload.

The way to locate the desired field and parse the data from MGEN is using a packet data parsing tool called Wireshark [24]. We capture the data from the emulator's transmitter using Wireshark, and analyse the data stream in the Wireshark's Graphical User Interface (GUI). We can choose any packet to see the data in it, Wireshark also processes the binary data in each field of headers and shows the corresponding values, which can be seen in Fig 2.5. It is convenient for us to parse the packets.

2.1.2 MGEN Command Line Usage

To configure the traffic flow generated from MGEN, we set the source, destination, protocol, pattern and other traffic flow options in the MGEN script text file. In this thesis, we configure the traffic flow transmitted into the down layer emulator by inputting the transmission events [13].



```
No.  Time         Source      Destination  Protocol  Length Info
5751 9.4536944…  10.99.0.3   224.1.2.8    UDP       370 45702 → 45702 Len=328
5752 9.4545515…  10.99.0.4   224.1.2.8    UDP       370 45702 → 45702 Len=328
57…  9.4567598…  10.99.0.5   224.1.2.8    UDP       370 45702 → 45702 Len=328
5754 9.4573527…  10.99.0.6   224.1.2.8    UDP       370 45702 → 45702 Len=328
5755 9.4597719…  10.99.0.7   224.1.2.8    UDP       370 45702 → 45702 Len=328
5756 9.4605229…  10.99.0.8   224.1.2.8    UDP       370 45702 → 45702 Len=328
5757 9.4625339…  10.99.0.9   224.1.2.8    UDP       370 45702 → 45702 Len=328
5758 9.4633882…  10.99.0.10  224.1.2.8    UDP       370 45702 → 45702 Len=328
5759 9.4657163…  10.99.0.1   224.1.2.8    UDP       370 45702 → 45702 Len=328

▸ Frame 5753: 370 bytes on wire (2960 bits), 370 bytes captured (2960 bits) on interface 0
▸ Ethernet II, Src: 02:00:05:01:00:01 (02:00:05:01:00:01), Dst: IPv4mcast_01:02:08 (01:00:5e:01:02:08)
▸ Internet Protocol Version 4, Src: 10.99.0.5, Dst: 224.1.2.8
▸ User Datagram Protocol, Src Port: 45702, Dst Port: 45702
▸ Data (328 bytes)

0000  00000001 00000000 01011110 00000001 00000010 00001000 00000010 00000000   ··^···..
0008  00000101 00000001 00000000 00000001 00001000 00000000 01000101 00000000   ······E·
0010  00000001 01100100 00011111 01100001 01000000 00000000 00000001 00010001   ·d·a@···
0018  01101100 10110111 00001010 01100011 00000000 00000101 11100000 00000001   l··c····
0020  00000010 00001000 10110010 10000110 10110010 10000110 00000001 01010000   ·······P
0028  11101101 11010010 00000000 00100101 00001000 00000101 00010000 11111111   ···%····
0030  11111111 00000011 00011000 10100010 10000011 00000010 00100010 00010000   ······"·
0038  01000001 10111001 00110010 01011100 10100010 00010001 01001110 00111001   A·2\··N9
0040  10011101 00100100 00001101 00110110 11000011 10010000 10111001 00111101   ·$·6···=
0048  00101010 00000111 00001000 00000000 00010000 00001001 00011000 10001111   *·······
0050  00000010 00000000 00000000 00000000 00000000 00000000 00000000 00000000   ········
0058  00000001 00011000 00000000 00000001 00000000 01101100 00000000 00000011   ·····l··
0060  00001010 00000001 00000101 00000000 00110111 00001000 00000111 00010000   ····7···
0068  00000111 00011000 11010000 10000001 00000001 00100000 11000000 10000100   ·····  ··
0070  00111101 00101000 11101000 11001000 11000000 11100011 11101000 10110101   =(······
```

Figure 2.5: Packet Parsing Window in Wireshark

Two significant commands are used in this thesis, they are pattern control and DATA assignment command line, `pattern [params]` and `DATA [<hex><hex>]`. Pattern command can change the arrival process distribution, we use this command to generate a Poisson arrival process. The arrival rate of the traffic flow can be set in the params field. Using DATA command can assign a sequence of hexadecimal values in the data payload.

2.1.3 MGEN Log File

There are two kinds of MGEN log files, one is mgen.log text file, the other one is $mgen.out$ text file. The mgen.log file help user debug when MGEN can't run correctly, this file show what the errors are and where they happen, which help user locate them.

The other log file is $mgen, out$, it contains all packets' information, such as the time when the transmission and reception event happens, the source and destination of each packet, the sequence number of each packet, the data payload saved in each packet and other important packet information, the transmission and reception absolute event time can be used to calculate the delay and lateness performance[25],

2.2 Introduction of EMANE Version 1.2.3

The Extendable Mobile Ad-Hoc Network Emulator (EMANE) [5] is an open source distributed network emulator, the emulator is modular and can be used with other tools and real hardware system. The emulator provides pluggable Media Access Control (MAC) and physical PHY layer, it supports developer to modify the original module and implement their design on it. EMANE can be integrated with upper layer emulators for real-time network experiments, such as MGEN [26] and Common Open Research Emulator (CORE) [27]. We use MGEN as the network upper layer event-generator, for MGEN is set as default traffic generator in EMANE's source code.

We run EMANE 1.2.5 on 64bits Ubuntu 14.04 installed on Virtual Box 5.2, we plug in a TDMA event scheduler radio model by downloading EMANE tutorial 8, all the radios, nodes and node's server address are set in the installation for the packets forwarding [28].

2.2.1 Installation of EMANE

EMANE uses a separate network IP stacks to emulate NEMs to get an independent emulation environment, Linux Containers (LXCs) is a good choice to implement it [29]. Because we are going to emulate several NEMs on one host computer and LXC method can be used to run multiple independent containers on one host LXC, we use Linux operating system in Ubuntu 14.04 installed on Oracle VM VirtualBox 5.02 which is a stable version to run EMANE.

Installation of EMANE is a complicated procedure, more than 30 pre-built packages need to be installed previously to support different modules run perfectly. Besides EMANE module, we also install 6 other apllications, they are MGEN, gpsd, olsrd, iperf, pynodestatviz, open-testpoint. Finally, we need to configure 10 nodes' and 10 radio's address to match the network topology. Even one lost step in installation causes system running problems, we have to check or reinstall from beginning which is a time-consuming procedure.

For convenience, we write a user installation guide which summarizes the procedure step by step. It help us in installation for the second time, the details in the guide remind us when we forget some main points in installation which saves us time. When the first time we install the emulator, we nearly spend 2 weeks in solving various bugs. But next time we transfer the emulator to another computer referring the guide, we just use 30 minutes. We also write a FAQ manual which includes some problems we frequently meet in the procedure, we share the manual with lab mates for future uses.

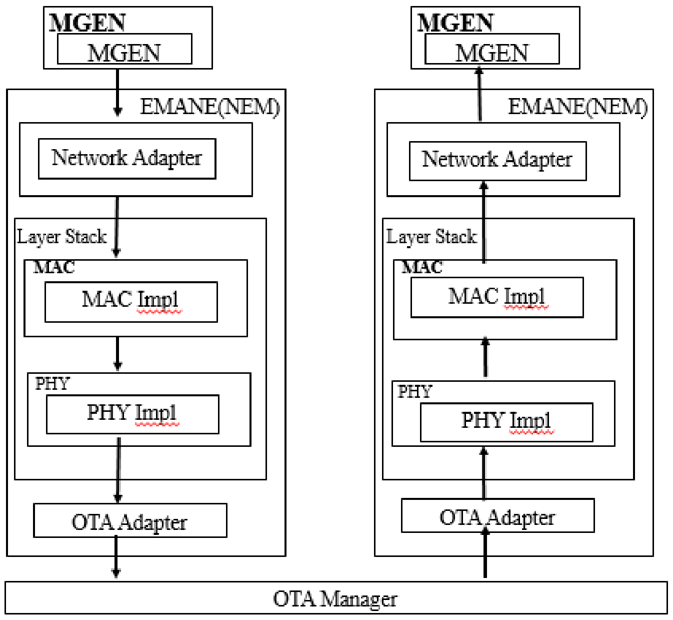
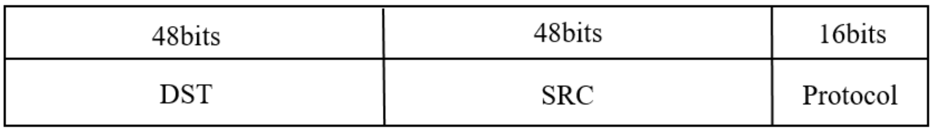


Figure 2.6: EMANE Architecture Generated from [5]

2.2.2 The Structure of EMANE

Fig 2.6 illustrates the architecture of EMANE, EMANE supports bidirectional multicast transmission. The downstream transmission process is shown in the following, MGEN generates UDP IP traffic flow into Network Emulation Modules (NEMs), the Network Adapter which is called virtual transport is the boundary of the NEMs, it provides a interface to read the traffic flow from upper layer application. The packets coming into the interface are encapsulated with a Ethernet header shown in Fig 2.7, each packet from the generator is assigned a MAC address by transferring the NEMs ID to MAC address, the method to perform it is writing the NEMs ID in last 16 bits in MAC address. By this mapping, the address forwarding becomes easy, all the addresses in EMANE are presented in NEMs ID. MAC implementation and PHY implementation emulates link layer, medium access and physical layer. The (over-the-air) OTA manager manages the data in unicast or multicast way, then the data is transmitted in over-the-air channel to other nodes. The upstream process is symmetric as the downstream process, NEM does the reverse operation as the downstream process, the traffic flow is received by MGEN in the end. We denote the transmitter NEM as EMANE node0 and denote the 10 NEMs as node1 to



48bits	48bits	16bits
DST	SRC	Protocol

Figure 2.7: Ethernet Header in EMANE Generated from [30]

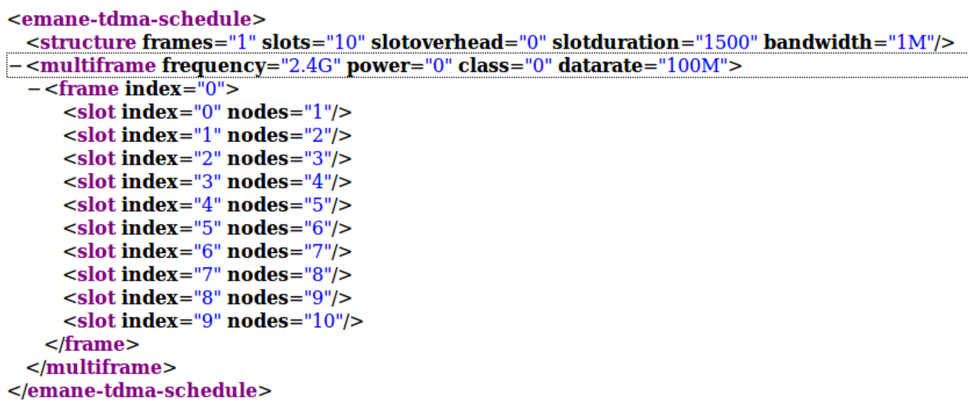
node10. We focus on the queuing system in EMANE's MAC layer and the TDMA scheduler radio model, which will be introduced in the next section.

2.2.3 TDMA Scheduler Model

We download the pluggable TDMA scheduler radio Model on `https://github.com/adjacentlink/emane-tutorial`, The TDMA transmission schedule scheme can be configured by the schedule XML file, each NEMs receiving events must have a schedule XML file. All the parameters related to TDMA scheduler module can be defined in this file, including slot size, slot overhead, number of slots and frames, slot data rate, bandwidth and so on [31]. The configuration of EMANE schedule XML file is in Fig 2.8, we set a fixed data rate in bit per second (bps), which means the service process is a deterministic process with a fixed service rate. The slot size in microseconds, we set 10 slots in one frame and the number of frame is one, slot is the minimum transmission unit contained in frame. We also set the 10 slots matching 10 different nodes, the index of slots matches the ID of nodes. The following command [31] is using TDMA schedule XML file to publish events.

```
[me@host 8]$ emaneevent−tdmaschedule schedule.xml −i emanenode0
```

EMANE TDMA schedule module supports the fragmentation when the packets size is larger than the transmit slot size, the packet is fragmented into two or more components fitted to the slot size, and the fragments will be assembled at the receiver. Whether to fragment packets can be set in EMANE. If the fragmentation is disabled, the large packet is dropped. We set the slot capacity large enough to transmit the packet, so EMANE doesn't fragment any packets, each transmit slot is assigned one packet at the TDMA radio model. The slot capacity



```
<emane-tdma-schedule>
  <structure frames="1" slots="10" slotoverhead="0" slotduration="1500" bandwidth="1M"/>
- <multiframe frequency="2.4G" power="0" class="0" datarate="100M">
  - <frame index="0">
      <slot index="0" nodes="1"/>
      <slot index="1" nodes="2"/>
      <slot index="2" nodes="3"/>
      <slot index="3" nodes="4"/>
      <slot index="4" nodes="5"/>
      <slot index="5" nodes="6"/>
      <slot index="6" nodes="7"/>
      <slot index="7" nodes="8"/>
      <slot index="8" nodes="9"/>
      <slot index="9" nodes="10"/>
    </frame>
  </multiframe>
</emane-tdma-schedule>
```

Figure 2.8: TDMA XML File

[26] is defined by

$$SlotCapacity = \frac{(SlotSize - SlotOverhead)}{\frac{1}{8} * 1000000 * ChannelDataRate}$$

2.2.4 EMANE Log File and debug

EMANE is a real time emulator without debug mode, the analysis on a real time emulator is limited [32]. As well, the compiling time is about 15 minutes on a 64GB RAM windows computer. Because EMANE is a real time emulator. The main debug way on EMANE is using $emane.log$ file, all the events happening in EMANE can be printed in the log file. The event information contains the event time, the layer where event happen, the node where event happens, the function name related to this event, the variable value related to this event and so on. We can print out all the values related to the area we want to debug and check the EMANE log file.

Chapter 3

Low Latency Queuing Control In Single-Hop Network

3.1 Model and Formulation

3.1.1 Queuing System Model

Our queuing system is a centralized single queue system with 10 single-hop servers, as shown in Fig. 3.1. The packets traffic flow starts in MGEN at time $t = 0$. The sequence of packets assigned due time in the application layer at local time instant $a_1, ..., a_n$ in MGEN, then we set the packets number n generated in traffic flow to be 10000, where $0 = a_1 \leq a_2 \leq ... \leq a_10000$. Each i-th arrived packet is assigned the destination address in its Ethernet header. To keep the queue in stable status, we set the maximum queue buffer size long enough as 1000. The reason why we set in this way is due to a tradeoff between queuing delay and queue overfilling. If we choose a long buffer size, the packets wait in a queue for a long time, queuing delay increases and influence our delay performance. If we choose a short buffer size, the buffer overfill soon with a high arrival rate, and queue come into a unstable status. After several tests, we finally choose a size 1000 for our buffer.

Because the original EMANE's service process is deterministic whose service rate μ = 70 packets per second, we configure the arrival process as a Markovian - Poisson process with a rate λ = 66 packets per second to increase the randomness and keep the system stable, "because an arrival from a Poisson process observes the system as if it was arriving at a random moment in time stated by the PASTA property [33]. "

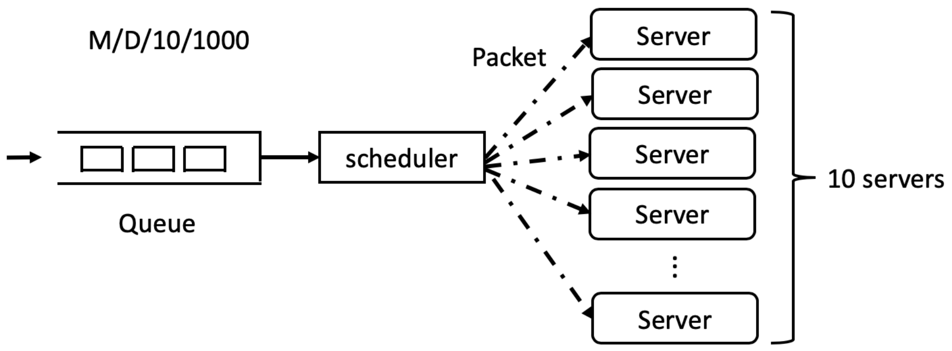


Figure 3.1: A Centralized Single Queue System with 10 Single-hop Servers

3.1.2 Scheduling Policy

EMANE's original queuing policy is First-Come, First-Served (FCFS), in this thesis, we add another queuing policy preemptive Earliest Due Date first (EDD) to observe the difference in latency reduction performance.

The policies we use in this thesis are all causal, which means our scheduling decisions is determined by the history and current system information. For the reason that service preemption is costly and possible to cause the problems in complexity and reliability [34], [35], we use non-preemptive policies in our queuing system, in which the process of a packet by a server must be finished before the server start to process another packet.

FIFO discipline is commonly used in electronic circuits for buffering and flow control between hardware and software [36]. FIFO is a method organizing the data in a buffer, the oldest data is processed firstly, and the newest data is processed in the end. FIFO queuing policy reduces the average delay of the queuing system [1]. C++ standard queue container use FIFO queue discipline [37] [38], the arriving element is inserted into the back position by enqueue operation, the queue removes the element at the front position by dequeue operation. The process is described in Fig 3.2. EDD Policy organize the data in a container according to the due time assigned to each packet, "the due time $d_i \in [0, \infty)$, also called due date, which is the time that i th packet is promised to be completed by the server [39], [1]." The due time is a soft deadline, we allow the packet finished processing after the time, but there will be a penalty happening. By this scheduling method, the lateness can be reduced [40].The due time setting is

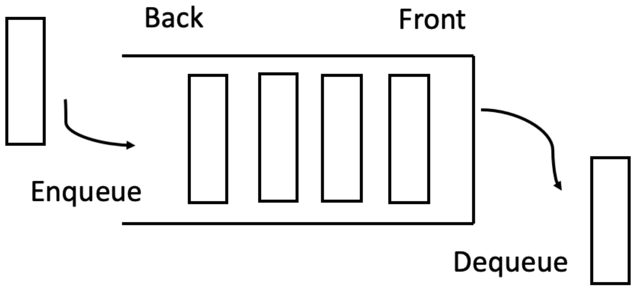


Figure 3.2: FIFO Queue Discipline Generated from [38]

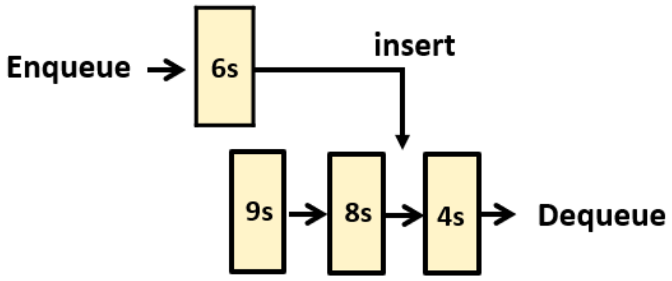


Figure 3.3: EDD Queue Discipline Generated from

related to the system time format, if we set all the time format in the system as a relative time, the due time is a relative time. If we want to use the due time in some emulator, the time format usually is absolute time format, because emulator uses the absolute computer local time for the real-time feature.

The elements container for EDD policy will change the order of data in queue according to their due time, the process is depicted in Fig 3.3. The earliest due time element is inserted to the beginning of the container from the time when the first packet arrives. The way EDD container find a right position to insert is that it compares the arrival element with the element in it from the beginning of the container until it finds a element whose due time is larger than the arrival element, then it inserts the arrival element in front of the larger due time one, and the arrival packets are inserted into the right position. The whole process is a sort process, it can be implemented by bubble sort and insert sort [11].

3.1.3 Delay Metrics

"Each packet i in the queue system is assigned a arrival time a_i and a completion time C_i, queuing delay is calculated by $D_i = C_i - a_i$, the lateness after the due time d_i is $L_i = C_i - d_i$, $T_i = \max[C_i - d_i, 0]$ is the tardiness (or positive lateness), and define the vectors including n elements $\boldsymbol{a}$= (a$_1, ..., a_n$), $\boldsymbol{d}$= (d$_1, ..., d_n$), $\boldsymbol{C}$= (C$_1, ..., C_n$), $\boldsymbol{D}$= (D$_1, ..., D_n$), $\boldsymbol{L}$= (L$_1, ..., L_n$) respectively [1]." In this thesis, we focus on the performance of jitter (lateness) probability, which presents the maximum lateness between two different policies.

We denote the scheduling policy as π, $\pi \in \{\boldsymbol{FCFS}, \boldsymbol{EDD}\}$. For any policy π, the **average delay** $D_{avg} : \mathbb{R}^n \to \mathbb{R}$ is defined by [1]

$$D_{avg}(\boldsymbol{C}(\pi)) = \frac{1}{n}\sum_{i=1}^{n}[C_i(\pi) - a_i]$$

The **maximum lateness** $L_{max} : \mathbb{R}^n \to \mathbb{R}$ is defined by [1]

$$L_{max}(\boldsymbol{C}(\pi)) = \max_{i=1,2,...,n} L_i(\pi) = \max_{i=1,2,...,n}[C_i(\pi) - d_i]$$

3.2 EMANE Queuing System Description

3.2.1 EMANE Priority Queues

The MAC layer queuing module is presented as the Fig 3.1, there are four priority queues in EMANE MAC layer. The module is defines in two C++ class, $BasicQueueManager$ and $Queue$. Packets in incoming traffic flow are sent to a queue based on its priority (0 to 4) [41], [31]. The priority used in EMANE is differentiated services code point (DSCP), as we introduced in Chapter 2, DSCP is a architecture to classify and manage packets flow [42]. DSCP occupies 6 bits of 8 bits type of service (TOS) [43], EMANE operates a 2 bits left shift on TOS value to get DSCP value. The mapping between DSCP and priority is illustrated in Fig 3.2, the meaning of DSCP values is presented in MGEN 4.02 user guide [13]. EMANE uses a default 0 TOS value for all packets which means the queue priority is locked to 0, all packets are assigned to the first queue according to the Fig 3.2. The slot assignment configuration is set in TDMA

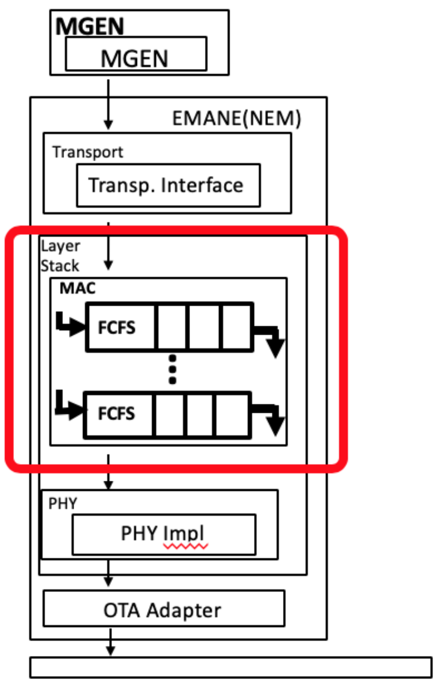


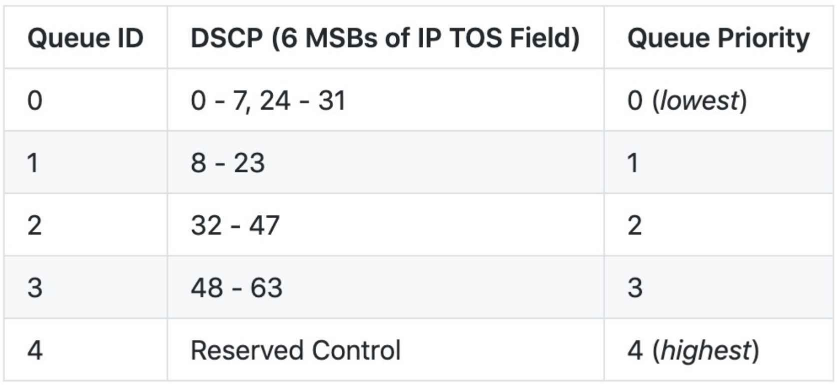
Figure 3.4: Queue Module in EMANE Generated from [5]

schedule XML file, each slot can be mapped with a priority queue. Whether the queue matching slot is strictly used for dequeue traffic is decided by parameter $queue.strictxdequeue$.

The priority queues' discipline in EMANE is FCFS, the order packets transmitted into slots is same as they are assigned to the queue. EAMEN assign each slot a matching NEM destination, packet will be transmitted when the packet matches the slot destination at the departure (dequeue) part[31].

3.2.2 Queue Configurations

The queue depth set in EMANE is 255, the queue depth is configured in a configuration file named $queue.depth$, this parameter controls all five priority queues. When packets overfill the queue, packets dropping happens. EMANE drops the packet at the beginning of the queue, which means the oldest packet in EMANE's queue module will be discarded. If fragmentation is allowed in EMANE, EMANE will drop the packet at the beginning of the queue regardless of it fragmentation status. Even parts of the packet which is divided into several fragments are transmitted, EMANE will also drop the oldest packet.



Queue ID	DSCP (6 MSBs of IP TOS Field)	Queue Priority
0	0 - 7, 24 - 31	0 (*lowest*)
1	8 - 23	1
2	32 - 47	2
3	48 - 63	3
4	Reserved Control	4 (*highest*)

Figure 3.5: Mapping Between DCSP and Queue Priority. Source [31]

3.3 Low Complexity MAC-Layer Queue Scheduling Module

Because original queue module uses FCFS policy which doesn't support our EDD algorithm, we design a Low complexity MAC-layer queue scheduling module which is shown in Fig 3.3, we implement a single EDD queue in scheduling module, EDD queue read the due time when packets arrives at the queue through an enqueue operation. The earliest due time packet is inserted to the beginning position of the queue and leaves queue by a dequeue operation when an idle TDMA slot call for the transmission. A congestion window receives the packet and transmit it to PHY layer.

3.3.1 EDD Algorithm in EMANE queuing system

The EDD queue module source code using C++ can be viewed in Appendix A-D. Our EDD queue module consists of class $NewBasicQueueManager$ and class $newqueue$ performs two functions, the function of class $NewBasicQueueManager$ is to drop the beginning packet in a full queue when new packet arrives, the function of class $newqueue$ is to insert the arriving packet at a appropriate position. The object of class $newqueue$ is declare in class $NewBasicQueueManager$, functions in class $newqueue$ are called by the $NewBasicQueue-$ $Manager$ object.

EDD algorithm in EMANE queuing system is described in Algorithm 1: When each of 10000 incoming packet arrives at the queue, queue module check whether its size reach the maximum limit. Queue module discard the packet at the beginning when the depth reaches

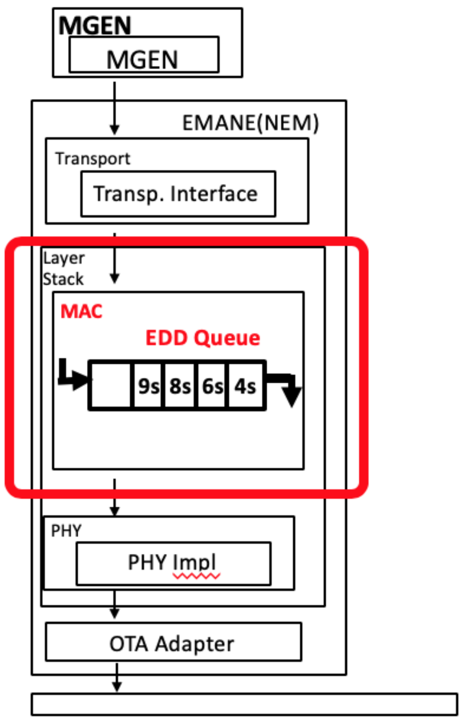


Figure 3.6: EDD Queue Module

maximum. Queue module read the due time assigned in application layer in each packet data payload. Then incoming packet is inserted the packet at the right position using module en-queue interface by the order of due time from earliest to latest. Queue module compares the incoming packet with the packet in the buffer from beginning. When the module find a packet whose due time is large than the incoming packet's, the queue module inserts the incoming packet behind that packet. If there is any server whose destination address matches the destination address of the earliest due date packet, the earliest due date packet is transmitted by the TDMA module using the dequeue interface to corresponding node.

3.3.2 Low Time Complexity List Sorting

Because the policy of current default queue module in EMANE is FCFS, we design a compared independent MAC-Layer queue module using EDD policy for jitter probability reduction. Because EMANE is a large scale emulator with high dependency, thousands of files call each other, the relation between the different files are hugely complex, just one small change cause

Algorithm 1 Earliest Due Date First (EDD)

```
1: $Q := \emptyset$;                    // $Q$ is the set of packets in the queue
2: while the system is ON and packet i arrives at the queue and packet number $n <= 10000$ do
3:     if $|Q| >= 1000$ then
4:         Delete the packet at the beginning of queue;
5:     end if
6:     $Q := Q \bigcup \{i\}$;
7:     while there are any idle servers do
8:         $j := \arg\min\{d_j : j \in Q\}$;        // $d_j$ is the due time of packet $j$
9:         Get the destination N of packet j
10:        if The Nth server is idle then
11:            $Q := Q/\{j\}$;
12:        end if
13:    end while
14: end while
```

lots of errors. As a result, we build a independent EDD queue module independent of EMANE original FCFS queue module.

Based on the EDD policy [1], [44], we should sort the elements in queue in an order from early due time to late due time, this is a sorting process. We compare two common sorting methods list sorting and bubble sorting [11], the time complexity of bubble sorting in finding is $\frac{n(n-1)}{2} = O(n^2)$. Compared with bubble sorting, the time complexity of list sorting in finding is just $n = O(n)$, which is significantly lower than bubble sorting.

As a result, we use C++ standard data container list [45] to implement the EDD queue module, which allows us to insert the packet at right position when packet is assigned to the queue.

3.4 Software Development by Code Migration

EMANE is a large scale emulator [46], once compiling time costs more than 15 minutes, which is inefficient for our software development. Furthermore, Modules in EMANE are highly coupled, one error or bug in syntax causes butterfly effect. The butterfly effect could causes ten or twenty errors in terminal and let the system run exceptionally, system crashes in the serious case. The efficiency of software development in EMANE is incredibly low.

To solve this problem, we move the queue module and its related code, such as packet module, congestion window module to other integrated development environment (IDE) Xcode

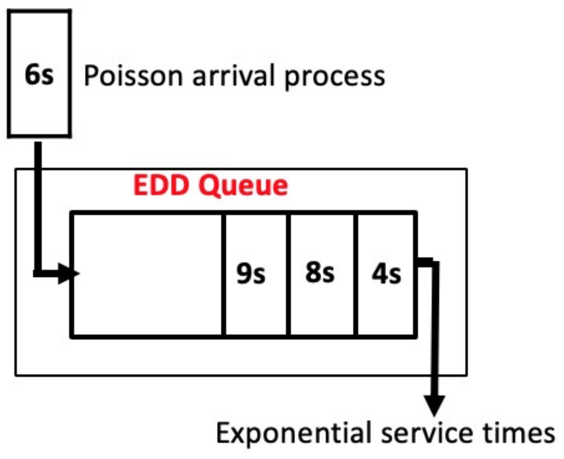


Figure 3.7: Debugging Environment outside of EMANE

with debug mode. In this IDE, we design and develop our EDD queue module entirely isolated from the original EMANE's queue mode to escape potential system exceptions. In the meanwhile, we simplify and delete some redundancy functions in original EMANE queue module, such as fragmentation for the TDMA slot bytes limit in the TDMA scheduling scheme is far larger than our packet's size.

3.5 Debug in an Outside Environment

EMANE is lack of debug mode, which is highly hard for our troubleshooting and packet tracking in queuing system. After our survey, we didn't find efficient debug and development method in current research. The common way to debug in EMANE is to print key variables into EMANE log file through a logging API [47]. All the variables are printed in log level to keep the most important information we need which helps us to troubleshoot. However, the log file includes all events' information where most of them are useless. The running time to record the information of more than 10000 packets is also long. The bigger problem is too many print statement consume so much memory that log file loses a large number of events information [48], which makes the log file data worthless. When we meet some bugs, EMANE doesn't create any information in debug level we need in the log file which prevents us from debugging. For these reasons, we come up with a new method for software development and debug in EMANE, that is code migration [49].

We build a debugging environment outside of EMANE described in Fig 3.4, we simplify original queue module and migrate the code into Xcode. In this environment, the higher layer is abstracted as Poisson arrivals. Low layer is abstracted as exponential services. We test our EDD queue module and check whether all the functions run correctly by using Xcode's debug mode [50]. After tests and debugging, we move EDD queue module back to EMANE. We match the enqueue and dequeue interface with EMANE's function $downstreampacketprocess()$ and $downstreampacketsend()$ respectively which calls these two interfaces. The little compilation time and convenient debug method in Xcode significantly reduce our debug time and improve debugging efficiency a lot.

Chapter 4

Emulation Results

In this section, we present the experiment results using this single hop low delay queuing system implemented on EMANE. We present some statistics to illustrate the low delay performance of two policies, FIFO and EDD policy. 1) We compare the jitter probability and queue length performance between two policies. 2) We also discuss the different queue performance when we set two different arrival rate $\lambda = 64$ packets per second and $\lambda = 66$ packets per second.

4.1 Test Configuration

The desktop PC for the testing is a ThinkStation P330 with a 3.2GHz Intel Core i7 8700 CPU having 8GB of 2666MHz DDR4 RAM and 1TB hard drive. We run a 64-bit Ubuntu 14.04 Linux with 50 GB disk. The MGEN 5.02 and EMANE 1.2.3 software are installed.

We configure the input system traffic flow by MGEN script, the arrival process is Poisson distribution, we test the system twice with different arrival rate $\lambda = 64$ packets per second and $\lambda = 66$ packets per second and same service process which is a deterministic process with $\mu = 70$ packets per second by defining a fix bit rate in EMANE TDMA schedule scheme, the arrival rate is strictly small than the departure rate, which means the system is in a stable status. The random arrival process Poisson rather than periodic process let queue size accumulate gradually, the latency gap between different policy is much more distinct [51]. The centralized queuing system consists of 10 single servers and we collect data at departure part to test the queuing lateness. We assume the arrival time of packet waiting in the queue is a_i in absolute time, the due time d_i of each packet is assigned with a_i and a_i added 8 microseconds with same probability $p = 0.5$, which is defined by

$$d_i = \begin{cases} a_i & \text{p=0.5} \\ a_i + 8 & \text{p=0.5} \end{cases}$$

To achieve an obvious performance in jitter probability, we set 10000 incoming packets with 512 Kilobytes. The queue module in EMANE drops packets when the queue is full, we set the queue size large enough as 1000 to avoid the packet loss, which causes unexpected queuing delay. The policy implemented are FCFS and EDD policy, we investigate the optimization in jitter by changing the order of packets transmitted in the buffer queue.

4.2 Data Extraction from EMANE Log File by Python Script

We collect the data in $emane.log$ using two Python substring crawler scripts to get lateness and queue length separately. By the lateness crawler script, we collect the lateness at 10 users of each packet under two different policy, which is used to plot the cumulative distribution function (CCDF) of jitter probability. All the lateness value are saved in a Python list data collection, and we write the data into a text file for future simulation on MATLAB.

With the queue length crawler script, we get the queue length before each packet arrives at the queue, and extract the event time. The time format in $emane.log$ introduced in the Chapter 2 is absolute time, to describe a relation between queue length versus buffering time, we need to transfer the event time in absolute time to relative time. We record the system start time, and set it as a reference time, and get the relative buffering time by several steps. We firstly dividing the event time in time point into 3 parts, hours, minutes and seconds with Python function $split(separator)$, this function divides one full string into several substrings saved in a list. We specify the separator as symbol : and we get three parts hours in integer, minutes in integer and seconds in decimal, because the type of list used to save substring is integer, we can't save the second part in decimal directly. In this reason, we secondly divide the second into two parts with a separator ".", and get the integer place $intsec$ and 6 digits decimal place

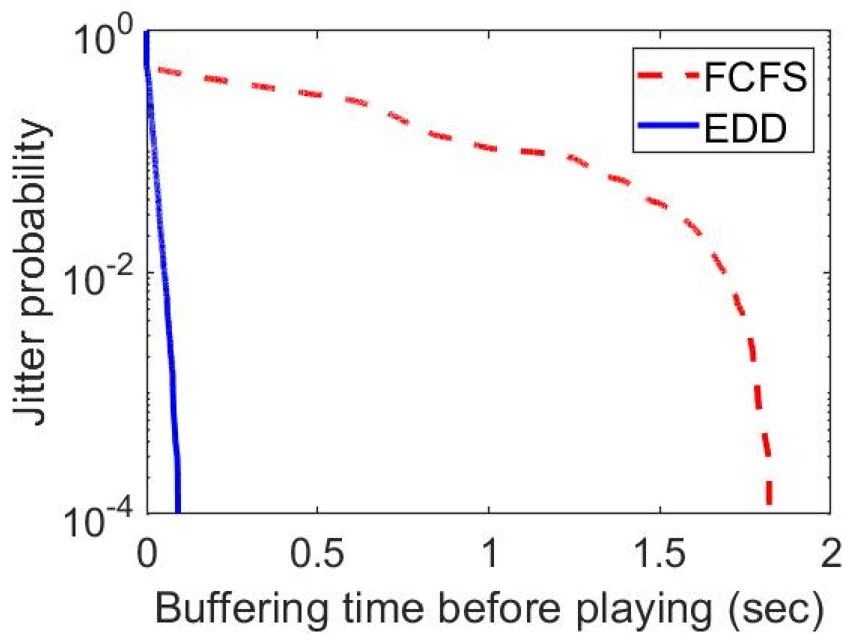


Figure 4.1: Complementary CDF of The Lateness Versus Buffering Time with $\lambda = 64$

$decisec$. Finally, we get the event relative time $relatime$ by

$$relatime = 3600 * h + 60 * m + intsec + 0.000001 * decisec$$

Because we set our system start time as the reference time, and get the buffering time by subtracting the start time from the event time.

We find Data lost problem sometimes appear in EMANE system which means it doesn't print all the information into the $emane.log$, although we write the print command in EMANE. The mismatch between buffering time versus queue length causes wrong performance in figures. We design the algorithm which let the crawler extract the buffering time and its matching queue length at the same time, this operation improve the accuracy of the simulation performance.

4.3 Lateness and Queue Length Performance with $\lambda = 64$

We use MATLAB to simulate the latency performance basing on the data collected from the $emane.log$ file. We assume EMANE transmits a video stream data, and lateness can be depicted as the buffering time before the video plays. Fig 4.1 illustrates the complementary CDF

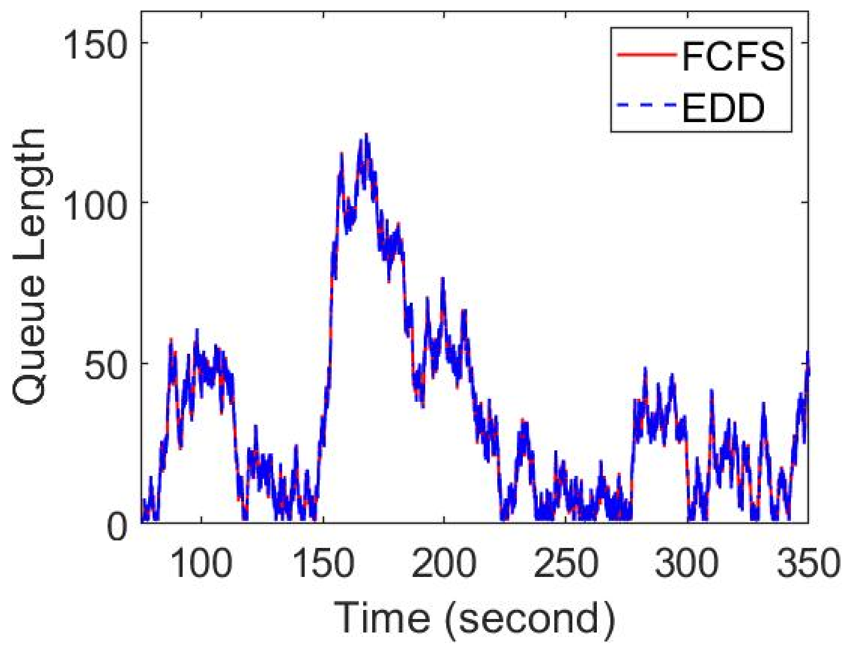


Figure 4.2: Queue Length Versus Time with $\lambda = 64$

of 10000 packets' lateness versus buffer time before playing with arrival rate $\lambda = 66$. The CCDF figure depicts the performance of the maximum lateness minimization between two policies clearly, as the packets in buffer grows, the lateness increase converges for we keep the system stable, we just need to compare the maximum latency value on the buffering time axis getting from the intersection between figure and the buffering time axis. We observed that our EDD module can achieve over 1000x reduction in jitter probability compared to the FCFS policy used in current EMANE module. The reason is that some packets that need to be sent immediately are sent several seconds later, FCFS module does not consider the priority in deadline.

Fig 4.2. shows the queue length versus time with arrival rate λ = 64. The number of incoming packets are more than 10000, considering the arrival rate, The simulation time is longer than 200 seconds, the long enough simulation time help us check the property in queue length and whether there is any error happening in system. The maximum queue size is less than 1000, which shows the simulation is under a stable status. The randomness of Poisson process make the queue length grows gradually, the queue length falls down to 0 after each rises is because the arrival rate is less than the departure rate. The figure also plots the queue

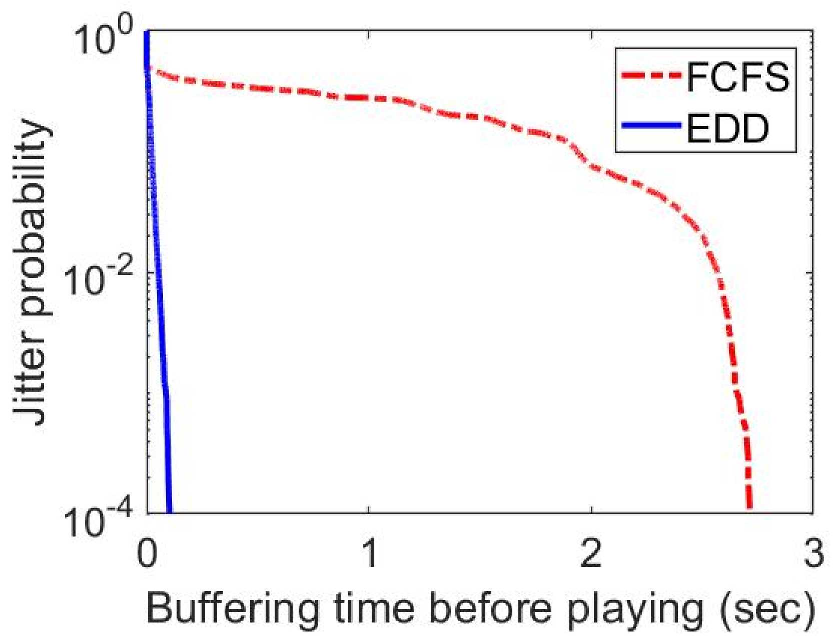


Figure 4.3: Complementary CDF of the Lateness Versus Buffering Time with $\lambda = 66$

length of EDD always keep pace with FIFO, that means packet reordering does not affect average delay when the arrival rate is fixed.

4.4 Lateness and Queue Length Performance with $\lambda = 66$

Fig 4.3 depicts the queue length versus time with arrival rate λ = 66. We can see the maximum buffering time of the two policies is larger than the buffering time in Fig 4.1. That because more packets arrive at the queue when arrival rate grows, the packet with a long due time needs to wait longer before it is transmitted.

Fig 4.4 presents the queue length versus time with arrival rate λ = 66. We can see the queue length of EDD always keep pace with FIFO's, and the average queue length with λ = 66 is obviously larger than with λ =64. By Little's formula [52],

$$E[Q] = \lambda E[D]$$

Where $E[Q]$ is the mean queue size, $E[D]$ is mean queuing delay and λ is the arrival rate. The equation shows the average size grows when λ increases, which explain the reason why the mean queue length with λ = 66 is larger than with λ =64.

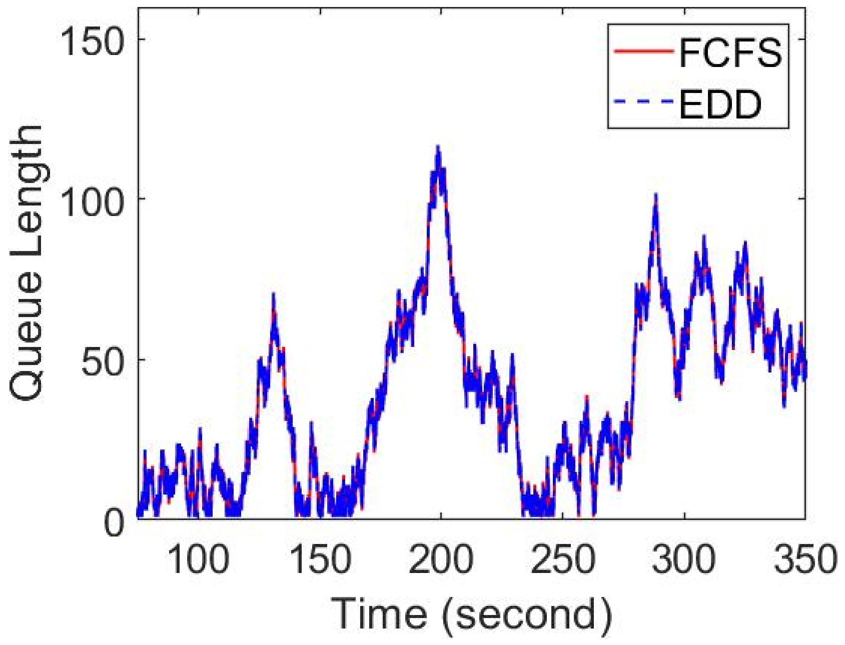


Figure 4.4: Queue Length Versus Time with $\lambda = 66$

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this paper, we present a design of low latency single hop queuing control system implemented on EMANE. We configure the incoming traffic flow by MGEN software and design our low complexity EDD queue module with C++ standard data structure list in EMANE. We also design an effective debug method for traffic flow bytes stream parsing and software development on EMANE. We use tool Wireshark to locate the specific data we need in the incoming bytes stream, and use code immigration along with platform transfer tests to improve the code and debug efficiency. We implement extensive experimental study with two different queue policy FCFS and EDD on our stable finite system and heterogeneous arrival and service process. The experimental results validated that EDD module in low latency queuing system achieve 1000x reduction in jitter probability compared to the FCFS module without effects on average delay.

5.2 Future Work

In this section, we propose the work we will do in the future.

5.2.1 Multiple Delay Objectives in Different Flows

We use a single flow queue in our current experiment, then we will expand the single flow to multiple flows with various queue policies to accomplish multiplexing communication. We need to consider how to identify packets in various policies and resolve the priority conflicts among different policies [53].

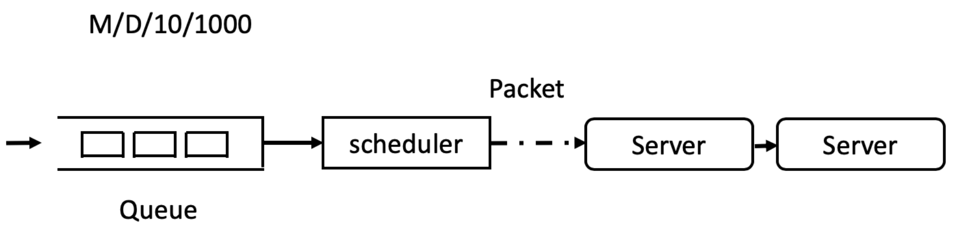


Figure 5.1: Multi-hop Queuing System

5.2.2 Low Latency Queuing Control in Multi-hop Network

In the future, We will change the present single-hop queuing system network topology to multi-hop system network topology, the network topology is shown is Fig 5.1. The multi-hop system consists of one transmitter, one relay and one receiver, they are all built by network emulation modules (NEMs) with the same protocol stack [7]. First node in figure is a transmitter, EDD queue module in transmitter process the downlink traffic flow and earliest due date packet is transmitted to the middle node firstly. The middle node performs as a bridge also with an EDD queue module, received traffic flow is received through PHY and MAC layer in the uplink path, then transmitted in a downlink path. In MAC layer, the earliest due date packet is forwarded to the third node firstly by the EDD module. We will change the current single-hop network to multi-hop network by using a topology constructed event file named emulation event log (EEL) file in EMANE [41].

Appendix A

NewBasicQueueManager.cc

```
#include "emane/models/tdma/NewBasicQueueManager.h"
#include "emane/models/tdma/newqueue.h"
#include <tuple>

EMANE::Models::TDMA::NewBasicQueueManager::NewBasicQueueManager():
newqueue_{}{}

EMANE::Models::TDMA::NewBasicQueueManager::~NewBasicQueueManager\\
(){}

size_t EMANE::Models::TDMA::NewBasicQueueManager::newenqueue\\
(DownstreamPacket new_pkt)
{
    size_t NewPacketDropped=0;
    auto ret = newqueue_.newenqueue(std::move(new_pkt));
        if(ret.second)
        NewPacketDropped = 1;
    return NewPacketDropped;
}

std::tuple<EMANE::Models::TDMA::NewMessageComponents,size_t>\\
EMANE::Models::TDMA::NewBasicQueueManager::newdequeue\\
(size_t requestedBytes)
{
    NewMessageComponents newcomponents;
    size_t newtotalLength = 0;
    auto newret = newqueue_.newdequeue(requestedBytes,true);
    size_t newLength = std::get<1>(newret);
    if(newLength)
    {
        newtotalLength+=newLength;
        auto & newparts = std::get<0>(newret);
        newcomponents.splice(newcomponents.end(),newparts);
    }
    return std::make_tuple(newcomponents,newtotalLength);
}
```

Appendix B

NewBasicQueueManager.h

```
#ifndef NewBasicQueueManager_hpp
#define NewBasicQueueManager_hpp

#include "emane/models/tdma/newmessagecomponent.h"
#include "emane/downstreampacket.h"
#include <tuple>
#include "emane/models/tdma/newqueue.h"

namespace EMANE
{
  namespace Models
  {
    namespace TDMA
    {
      class NewBasicQueueManager
      {
      public:
        NewBasicQueueManager();
        ~NewBasicQueueManager();
        NewQueue newqueue_;
        size_t newenqueue(DownstreamPacket new_pkt);
        std::tuple<EMANE::Models::TDMA::NewMessageComponents,\\
        size_t>newdequeue(size_t requestedBytes);
```

```
            };
         }
      }
   }

#endif
```

Appendix C

newqueue.cc

```
#include "emane/models/tdma/newqueue.h"


EMANE::Models::TDMA::NewQueue::NewQueue(){}

EMANE::Models::TDMA::NewQueue::~NewQueue(){}


std::pair<EMANE::DownstreamPacket,bool>\\
EMANE::Models::TDMA::NewQueue::newenqueue(DownstreamPacket new_pkt)
{
    DownstreamPacket NewDroppedPacket({{},{},{},{}},{},{});
    bool DroppedPacketFlag{0};
    if(New_List.size() == 255)
    {
        auto newentry = New_List.begin();
        DroppedPacketFlag = true;
        NewDroppedPacket = New_List.front();
        New_List.erase(newentry);
    }
    pkt_iterator = New_List.begin();
    while(pkt_iterator->getPacketInfo().DueTime_ <\\
    new_pkt.getPacketInfo().DueTime_ && pkt_iterator!= New_List.end())
    ++pkt_iterator;
    New_List.insert(pkt_iterator, new_pkt);
    return {std::move(NewDroppedPacket),DroppedPacketFlag};

}
```

```
std::tuple<EMANE::Models::TDMA::NewMessageComponents,size_t,std::list\\
<EMANE::DownstreamPacket>>EMANE::Models::TDMA::NewQueue::newdequeue\\
(size_t requestedBytes, bool bDrop)
{
    NewMessageComponents components{};
    size_t newtotalBytes = 0;
    std::list<EMANE::DownstreamPacket> dropped{};
     while(newtotalBytes < requestedBytes)
    {
        if(!New_List.empty())
        {
             auto DeQueuePacket = New_List.begin();
             NEMId dst{DeQueuePacket->getPacketInfo().getDestination()};
             size_t duti{DeQueuePacket->getPacketInfo().getDueTime()};
             Priority prio{DeQueuePacket->getPacketInfo().getPriority()};
             Utils::VectorIO vect{DeQueuePacket->getVectorIO()};
             size_t PacId{DeQueuePacket->getPacketInfo().getPacketId()};
             TimePoint EDDaritime{DeQueuePacket->getPacketInfo().\\
             getEDDArrivalTime()};
             TimePoint FIFOaritime{DeQueuePacket->getPacketInfo().\\
             getFIFOArrivalTime()};
             components.push_back({dst,duti,prio,vect,PacId,EDDaritime,\\
             FIFOaritime});
             newtotalBytes += DeQueuePacket->length();
             New_List.erase(DeQueuePacket);
             break;
         }
         else
         {
             break;
         }
    }
    return std::make_tuple(components,newtotalBytes,std::move(dropped));
}
```

Appendix D

newqueue.h

```
#ifndef New_Queue_hpp
#define New_Queue_hpp

#include "emane/component.h"
#include "emane/downstreampacket.h"
#include "emane/models/tdma/messagecomponent.h"
#include "emane/models/tdma/newmessagecomponent.h"

namespace EMANE
{
  namespace Models
  {
    namespace TDMA
    {
      class NewQueue
      {
      public:
        NewQueue();
        ˜NewQueue();
        std::pair<DownstreamPacket,bool>\\
        newenqueue(DownstreamPacket new_pkt);
        using NewPacketQueue = std::list<DownstreamPacket>;
        NewPacketQueue New_List;
        std::tuple<NewMessageComponents,size_t,std::list\\
        <DownstreamPacket>>newdequeue(size_t requestedBytes, bool bDrop);
    private:
        NewPacketQueue::iterator pkt_iterator;
      };
    }
  }
}
#endif
```

Reference

[1] Yin Sun, C Emre Koksal, and Ness B Shroff. "Near delay-optimal scheduling of batch jobs in multi-server systems". In: *Ohio State Univ., Tech. Rep* (2017).

[2] Yin Sun, C Emre Koksal, and Ness B Shroff. "On delay-optimal scheduling in queueing systems with replications". In: *arXiv preprint arXiv:1603.07322* (2016).

[3] Glenn Judd et al. "Using physical layer emulation to optimize and evaluate mobile and wireless systems". In: *Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*. 2008, p. 26.

[4] Alberto Alvarez et al. "Limitations of network emulation with single-machine and distributed ns-3". In: *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*. ICST (Institute for Computer Sciences, Social-Informatics and ... 2010, p. 67.

[5] LLC Adjacent Link. *Extendable Mobile Ad-hoc Network Emulator (EMANE)*. `https://www.nrl.navy.mil/itd/ncs/products/emane`.

[6] Matija Pužar and Thomas Plagemann. "NEMAN: A network emulator for mobile ad-hoc networks". In: *Research report http://urn. nb. no/URN: NBN: no-35645* (2005).

[7] Kaustubh Jain et al. "Studying real-time traffic in multi-hop networks using the EMANE emulator: capabilities and limitations". In: *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*. ICST (Institute for Computer Sciences, Social-Informatics and ... 2011, pp. 93–95.

[8] *NS-3. Network simulator 3*. `https://www.nsnam.org/`.

[9] *Common Open Research Emulator (CORE)*. https://www.nrl.navy.mil/itd/ncs/products/core.

[10] *RoboCom*. http://www.robocomtech.com/network-emulation-vs-simulation/.

[11] Sardar Zafar, Abdul Wahab, et al. “A new friends sort algorithm”. In: *2009 2nd IEEE International Conference on Computer Science and Information Technology*. IEEE. 2009, pp. 326–329.

[12] Naval Research Laboratory (NRL) PROTocol Engineering Advanced Networking (PROTEAN) Research Group. *MGEN*. https://www.nrl.navy.mil/itd/ncs/products/mgen.

[13] *MGEN user guide version 5.02*. https://downloads.pf.itd.nrl.navy.mil/docs/mgen/mgen.html#Command-line_Options.

[14] Clement Kam, Sastry Kompella, and Anthony Ephremides. “Experimental evaluation of the age of information via emulation”. In: *MILCOM 2015-2015 IEEE Military Communications Conference*. IEEE. 2015, pp. 1070–1075.

[15] *OSI model*. https://en.wikipedia.org/wiki/OSI_model.

[16] *IPv4 Header*. https://en.wikipedia.org/wiki/IPv4#Header.

[17] William Stallings. “Data and Computer Communications tenth edition”. In: 2014.

[18] *Network byte order and host byte order*. https://www.ibm.com/support/knowledgecenter/en/SSB27U_6.4.0/com.ibm.zvm.v640.kiml0/asonetw.htm.

[19] *ntohs(3) - Linux man page*. https://linux.die.net/man/3/ntohs.

[20] David Clark. *IP datagram reassembly algorithms*. Tech. rep. RFC 815, July, 1982.

[21] *List of IP protocol numbers*. https://en.wikipedia.org/wiki/List_of_IP_protocol_numbers.

[22] *User Datagram Protocol*. https://en.wikipedia.org/wiki/User_Datagram_Protocol.

[23] *List of TCP and UDP port numbers*. `https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers#Well-known_ports`.

[24] Kaustubh Jain et al. *Study of OLSR for Real-time Media Streaming over 802.11 Wireless Network in Software Emulation Environment*. Tech. rep. 2010.

[25] Bala Chidambaram and Yueli Yang. "Cyber Analysis System Toolkit: A high-fidelity, virtual cyber test-bed for network modeling and experimentation". In: *MILCOM 2012-2012 IEEE Military Communications Conference*. IEEE. 2012, pp. 1–5.

[26] *Distributed wireless network emulation framework on Github*. `https://github.com/adjacentlink/emane`.

[27] Jeff Ahrenholz, Tom Goff, and Brian Adamson. "Integration of the CORE and EMANE Network Emulators". In: *2011-MILCOM 2011 Military Communications Conference*. IEEE. 2011, pp. 1870–1875.

[28] Ryan Spangler. "Packet sniffing on layer 2 switched local area networks". In: *Packet-watch Research* (2003), pp. 1–5.

[29] *EMANE Networking Waveform Emulation*. `https://www.robocomtech.com/emane-network-emulation/`.

[30] *Ethernet frame*. `https://en.wikipedia.org/wiki/Ethernet_frame#Header`.

[31] *Github Wiki for TDMA Model*. `https://github.com/adjacentlink/emane/wiki/TDMA-Model#tdma-schedule`.

[32] Andrea L Brennen et al. "Worth a thousand bits: visual encoding of tactical communication network data". In: *MILCOM 2013-2013 IEEE Military Communications Conference*. IEEE. 2013, pp. 1334–1340.

[33] Françis Baccelli et al. "The role of PASTA in network measurement". In: *IEEE/ACM Transactions on Networking (TON)* 17.4 (2009), pp. 1340–1353.

[34] Kay Ousterhout et al. “Sparrow: distributed, low latency scheduling”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 69–84.

[35] Abhishek Verma et al. “Large-scale cluster management at Google with Borg”. In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM. 2015, p. 18.

[36] *FIFO (computing and electronics)*. `https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)`.

[37] William E Woods, Philip E Stanley, and Thomas S Hirsch. *Queue structure for a data processing system*. US Patent 4,320,455. Mar. 1982.

[38] *Std::queue in Cplusplus*. `http://www.cplusplus.com/reference/queue/queue/)`.

[39] Michael Pinedo and Khosrow Hadavi. “Scheduling: theory, algorithms and systems development”. In: *Operations Research Proceedings 1991*. Springer, 1992, pp. 35–42.

[40] James R Jackson. “Scheduling a production line to minimize maximum tardiness”. In: *management science research project* (1955).

[41] Alexandre Nikodemski et al. “Reproducing measured MANET radio performances using the EMANE framework”. In: *IEEE Communications Magazine* 56.10 (2018), pp. 151–155.

[42] *Differentiated Services*. `https://en.wikipedia.org/wiki/Differentiated_services`.

[43] Fred Baker et al. *Definition of the Differentiated Services Field (DS field) in the IPv4 and IPv6 headers*. Tech. rep. RFC 2474, Dec, 1998.

[44] D Towsley and SS Panwar. “On the optimality of minimum laxity and earliest deadline scheduling for real-time multiprocessors”. In: *Proceedings. EUROMICRO’90 Workshop on Real Time*. IEEE. 1990, pp. 17–24.

[45] *Std::list in Cplusplus*. `http://www.cplusplus.com/reference/list/list/?kw=list)`.

[46] Leonid Veytser, Bow-Nan Cheng, and Randy Charland. "Integrating radio-to-router protocols into EMANE". In: *MILCOM 2012-2012 IEEE Military Communications Conference*. IEEE. 2012, pp. 1–6.

[47] *Logging Services Manual*. `http://logging.apache.org/log4j/1.2/manual.html`.

[48] *Data Storage Memory Management: Optimizing Computer Memory*. `https://www.enterprisestorageforum.com/storage-hardware/memory-management.html`.

[49] Kostas Kontogiannis et al. "Code migration through transformations: An experience report". In: *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press. 1998, p. 13.

[50] Apple. *Debugging with Xcode*. `https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/debugging_with_xcode/chapters/debugging_tools.html`.

[51] University of Bristol. "Simple queueing models". In: 2012.

[52] Moshe Zukerman. "Introduction to queueing theory and stochastic teletraffic models". In: *arXiv preprint arXiv:1307.2968* (2013).

[53] Drew Bertagna. *Qualified priority queue scheduler*. US Patent 6,934,294. Aug. 2005.