Networking support for xv6

Operating Systems - CS3500 (July-Nov '20)

CS18B055 - E. Sai Dheeraj
CS18B046 - Rushabh Lalwani

Prof. Chester Rebeiro

INTRODUCTION

In this project, we have written a device driver support for a network interface card (NIC) and also added support for UDP network sockets to xv6. Some important terms to be known before are:

QEMU: It is an open-source machine emulator or virtualiser and provides a lot of hardware abstraction/virtualization. xv6 OS will run as a host on it.

NIC: Network Interface Card is a hardware component required for network communication e.g E1000. NIC driver is a software solution that translates the NIC language to OS.

PCI: Peripheral Component Interconnect is a hardware bus used to connect internal components of the system.

E1000 virtual driver: software emulation of E1000 card provided by QEMU.

TAP: a networking backend which offers very good performance and can be configured to create virtually any type of network topology.

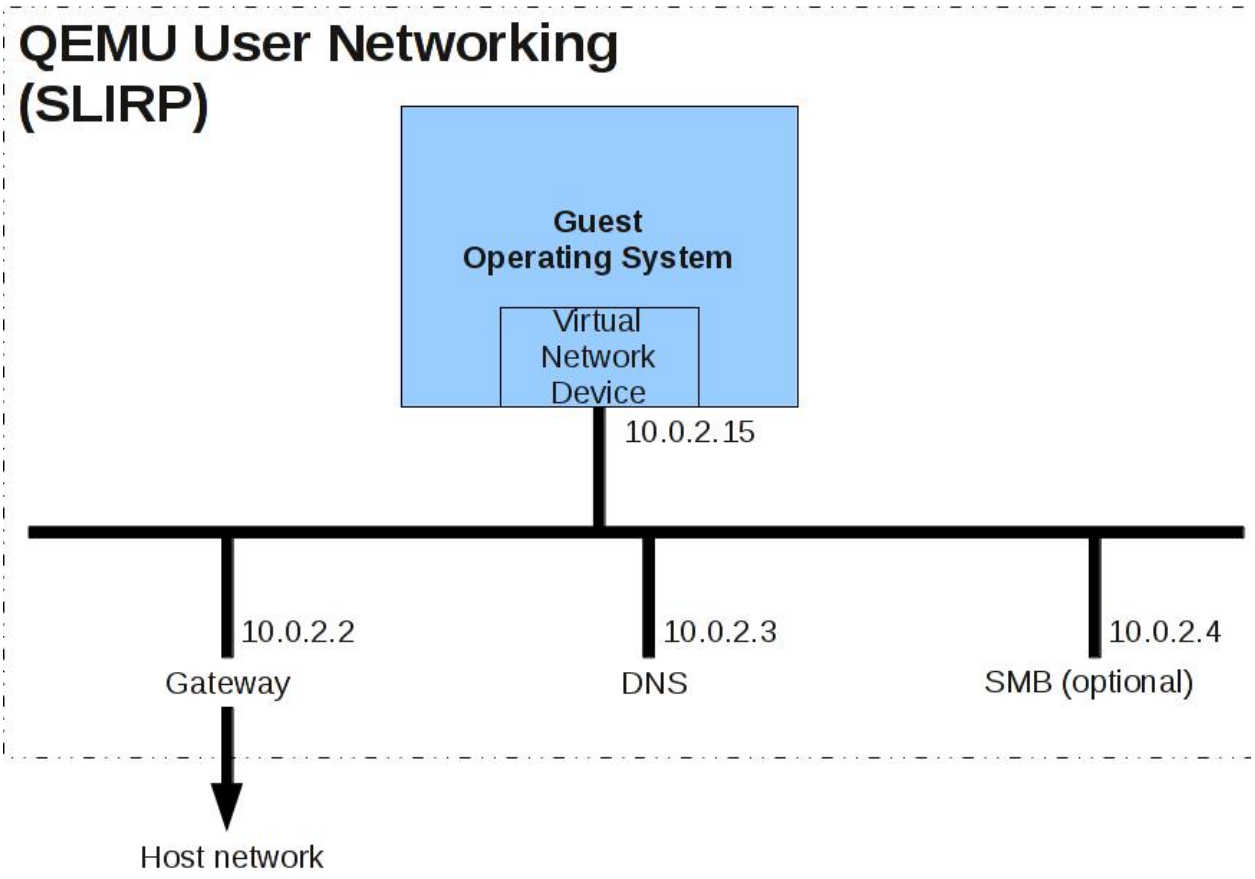
UDP: User Datagram protocol. It is a transport layer and connectionless protocol and more efficient in terms of both latency and bandwidth compared to TCP.

We have cloned the **xv6 repository**[7] following the Lab 10 of MIT's Operating System course - **6.S081** and explained the initial changes made by their team and further work done by us. We have also followed their webpage[1].

BACKGROUND

QEMU SETUP

QEMU can simulate several virtual network devices/cards for the guest host (xv6) to handle network communication. We will be using the PCI E1000 card which QEMU emulates and gives a hardware impression connected to real Ethernet LAN. In reality, the E1000 driver will interact with an emulation (ip: 10.0.2.2) provided by QEMU, connected to a LAN that is also emulated by QEMU. This emulation also behaves as a firewall. Note that we **have not added support** for DNS and SMB in this project. Refer this image from here[6]:



On the LAN, xv6 guest has an ip address 10.0.2.15. The support is added in Makefile:

```
QEMUOPTS += -device e1000, netdev=net0, bus=pcie.0
```

QEMU connects these drivers with the network backend on our host by using user-mode network stack (or TAP devices). We will use this unprivileged network stack to interact with the emulated NIC. The support is added in Makefile:

```
QEMUOPTS += -netdev user, id=net0, file=packets.pcap
```

All the incoming and outgoing packets are recorded in *packets.pcap* file in the xv6 directory. The messages can be read from the file using the command:

```
$ tcpdump -XXnr packets.pcap
```

When QEMU is booted, using above command, we get the output:

```
reading from file packets.pcap, link-type EN10MB (Ethernet)
```

net.h net.c

Net.h file is the header file for several data structures we will be using for networking. It includes the following data structures:

1. mbuf (used as packet buffer)
2. mbufq (queue of mbuf)
3. ip (IP packet header)
4. eth (Ethernet packet header)
5. udp (UDP packet header)

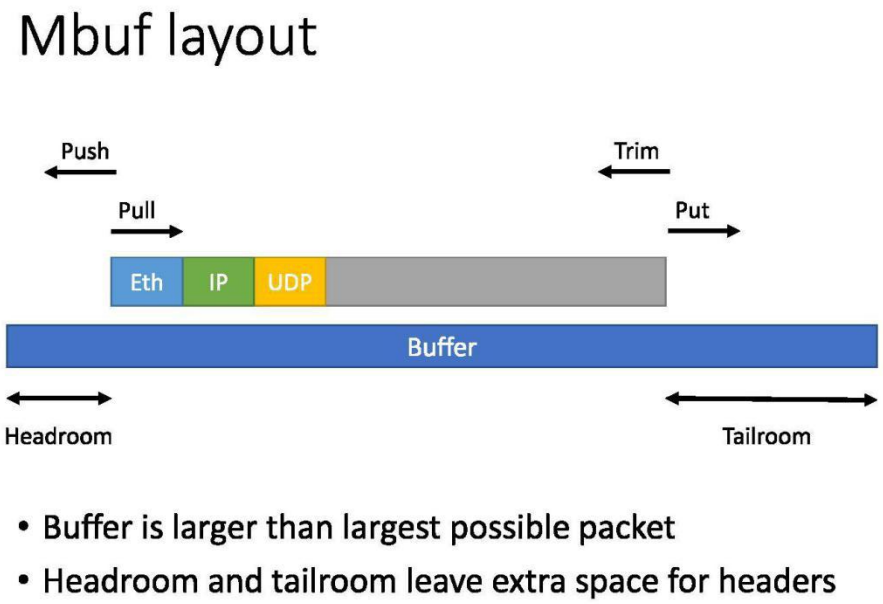
It also includes signatures for all the methods and functions of mbuf and mbufq and macros for a few of them.

Net.c file contains all the functions and methods corresponding to the above structures:

1. mbuf structure has a buffer *buf* of size *MBUF_SIZE* (=2048), an integer *len* to track the length of the buffer used and a pointer *head* to track the head of the buffer. The functions *mbufalloc*() and *mbuffree*() behave structures' constructor and destructor.

2. The functions (or methods) *mbufpull*(), *mbufpush*(), *mbufput*(), *mbuftrim*() are used to modify the length and head pointer of the buffer.
3. mbufq structure has a head and tail pointer and hence can behave as a queue. To provide functionality like a queue, it has methods like *mbufq_pushtail*(), *mbufq_pophead*(),

mbufq_em
image[3]:



4. To send UDP packets, we use the following functions sequentially. *net_tx_udp* is called from *sockwrite*() function which is defined in Network Sockets.
    a. net_tx_udp (sends an UDP packet)
    b. net_tx_ip (sends an IP packet)
    c. net_tx_eth (sends an ethernet packet)
    d. e1000_transmit (defined later)

5. To receive a packet, we use the following functions sequentially:
    a. net_rx (called by interrupt driver to deliver packet)
    b. net_rx_ip (receives an IP packet)
    c. net_rx_udp (receives a UDP packet)
    d. sockrecvudp (defined later)

NETWORK DEVICE DRIVER

In this part we will complete the implementation of E1000 Device Driver with the help of this manual[2]. The functions which will discover and initialize the device, and handle interrupts (which occurs only when the device is receiving packets). The functions for transmitting and receiving the packets are written.

Both sending and receiving packets is managed by a separate queue of descriptors that is shared between xv6 and the E1000 in memory.

TRANSMISSION

- Mbuf linked list is continuous packets which is handled by a circular queue (ring), whichever descriptor is currently free (usually the tail descriptor) will be given the task of transmitting the packet (which is a character string defined as mbuf structure).
- The hardware takes the responsibility of changing the head pointer of the ring once the task of the descriptor is done (even if the packet is not delivered).
- We have handled the case where the ring could be overflowing , i.e., the ring is full and none of the descriptors are given back by the hardware.
- While assigning the buffer data to the descriptor, we set the **cmd** (command field) flag to **RS** (to get report status - used for debugging) and **EOP** (end of packet - so that multiple descriptors can be used to form a packet).

RECEIVING

- Receiving packet is identified using the Absolute timer (**RDTR**) by the hardware.
- This is also similar to the Transmission in terms of using the ring of descriptors, the tail descriptor is used every time we want to receive a packet.
- There is no need of taking care of the case where all descriptors in the ring are used because the timer interrupt which occurs every time we have to receive a packet takes care that this situation doesn't occur.
- We can deliver the mbuf to the protocol layer using *net_rx()* function, the *e1000_init()* function which is called inside this function allocates the mbufs to each slot of receive ring and we have to run a while loop for this reason in the receive function that we have implemented.

NETWORK SOCKETS

Network sockets are a standard abstraction for OS networking that bear similarity to files. Sockets are accessed through ordinary file descriptors (just like pipes, inodes and devices). Reading from a socket file descriptor receives a packet while writing to it sends a packet.

As we have finished the E1000 driver, we can support user applications to perform network communication. To interact with the xv6 we will use sockets. We will implement a stripped down version of sockets that supports UDP.

Each network socket only receives packets for a particular combination of local and remote IP addresses and port numbers, and xv6 is required to support multiple sockets. A socket can be created and bound to the requested addresses and ports via the *sys_connect* system call, which

returns a file descriptor. This system call is implemented in *kernel/sysfile.c.*

We define a data structure for socket and its related methods in *kernel/sysnet.c*. As socket and pipes are both types of file, we referred to the implementation of pipes in *kernel/pipe.c*. A socket will contain a queue of memory buffers (**mbufq**) to store the packets waiting to be received, remote and local IP addresses and port numbers. Received packets will stay in this queue until the *read*() system call dequeues them. We maintain a singly linked list of all active sockets called ***sockets***. It is useful for finding which socket to deliver newly received packets to.

socketalloc() function initializes a file to socket type, allocates mbuf and other and adds this newly created socket into the *sockets* list. We will implement four functions which are support handler (called from *kernel/net.c*) and reading, writing and closing sockets (invoked correspondingly from *kernel/file.c*).

sockclose()

- The socket is cleared with the help of this function. We iterate through the sockets and check which socket matches in the list.
- After finding the socket, we free all the mbufs' which are not read before (which are present in the queue).
- Finally, we free the socket object.

sockread()

- This function is used to read a mbuf present in the given socket and send this packet to user space.
- We first check if the mbufq is empty by using the *mbufq_empty*() function.
- If the queue is empty, we sleep to wait inside a while loop until it gets enqueued and non-empty. In case the process gets killed, we break and stop.
- Now we pop the first element of mbufq queue using *mbufq_pophead*() and use *copyout*() to move its payload data in user space at the given address.
- Finally we free the mbuf.

sockwrite()

- This function is used to create a mbuf for a given socket, allocate payload from user space and send this UDP packet.
- We first allocate a new mbuf by giving headroom for local and remote ip address and port number.

- After successful allocation, we append the size of data we need to payload using *mbufput*().
- Then we use *copyin*() to copy the payload from user space by providing the address and size.
- Finally we send the packet using *net_tx_udp*() function.

sockrecvudp()

- This function is called by the protocol handler layer (*net_rx_udp*()) to deliver UDP packets.
- The function first iterates through *sockets* to match the socket's ip addresses and port numbers. If it fails to find the corresponding socket, the mbuf is not delivered and hence free'd.
- After finding the corresponding socket, the packet or mbuf is pushed into the socket's mbuf queue (*mbufq*).
- Now that the socket's *mbufq* is non-empty, we wake up the queue which might be potentially sleeping.

RUNNING TOGETHER

The program *nettests.c* in the user folder will help test both the parts together. This testing file will send UDP packets to the localhost outside QEMU and receive packets from it.

- Whenever we try to send a packet to some IP address, our system will first contact DNS (Domain Name service) which will let us know where the destination host is i.e., whether the destination host IP is in the same network (in our case) or in a different network.
- One *ping* means sending a packet to the destination server and receiving a packet in return.
- The testing is done in multiple levels
    1. Testing a single ping.
    2. Doing multiple pings from a single process.
    3. Running a *for* loop and doing *fork()* inside to create multiple child processes and doing ping from these multiple processes.
- In order to check the code , run "make server" on one terminal and "make qemu" and then "nettests" on another terminal. We see "Hello world!" getting printed in the first terminal and lots of "OK" and all test cases passed in the second terminal.

CONCLUSION

We successfully exploited hardware abstractions provided by QEMU for sending and receiving UDP packets on the E1000 PCI virtual driver. UDP is a simple network protocol which works very fast for sending and transmitting data packets. QEMU allows the flexibility to use a completely user

mode network stack which requires no administrative privileges but has a lot of overheads.

REFERENCES

1. https://pdos.csail.mit.edu/6.S081/2019/labs/net.html
2. https://pdos.csail.mit.edu/6.S081/2019/readings/hardware/8254x_GBe_SDM.pdf
3. https://pdos.csail.mit.edu/6.S081/2019/lec/l-networking.pdf
4. https://pdos.csail.mit.edu/6.S081/2019/xv6/book-riscv-rev0.pdf
5. https://www.qemu.org/docs/master/system/net.html
6. https://wiki.qemu.org/Documentation/Networking#Tap
7. The github repo : https://github.com/mit-pdos/xv6-riscv-fall19