PKU-CompNet (H) Fall'21
Lab Assignment L2: Protocol Stack

Contents

0 Due

- Part A (Section 3.2 Link-layer): Oct. 20, 2021 (23:59:59 PM)
- Part B (Section 3.3 Network-layer): Nov. 3, 2021 (23:59:59 PM)
- Part C (Final submission): Nov. 29, 2021 (23:59:59 PM)

1 Introduction

In this lab, we are going to implement a userspace protocol stack (including Layer 2, 3, and 4) based on `libpcap` [man page: `http://www.tcpdump.org/manpages/pcap.3pcap.html`].

After finishing this lab, you are expected to:

- Deeply understand:
  - Classic TCP/IP standards and their implementation
  - How to implement a usable system by programming
- Be (more) familiar with:
  - Bash scripts
  - Ethernet frames
  - `man` pages and RFCs
  - Usage of Linux network tools

2 Toolkit

- Create Virtual Networks

  It's naturally impossible for us to test our implementation in a large real network: just because we don't have that many computers. For network system designers, it's often essential to create some sort of virtual networks for developing and testing. Here we present a home-made tool, `vnetUtils`, to create and use virtual networks with specified topology, with the constraint that there are no redundant links.

  `vnetUtils` is a set of small bash scripts, providing the functionalities of creating virtual hosts, connecting them and running commands on them. with the help of it you can easily develop scripts to create virtual networks with desired topology. See its `README.md` for details.

- Monitor the Network

  When you are developing the protocol stack, you may find it very useful to see how the (virtual) network reacts to your program at all times. We recommend you to make use of `Wireshark` or `tcpdump`, which lets you see what's happening on your network at a microscopic level. Go to the website for download and detailed usage.

3 Instructions

All the following tasks are marked any one of **PT (Programming Task)**, **WT (Writing Task)**, **CP (Checkpoint)**, or **CL (Challenge)**. For the programming tasks, you should submit your implementation (additional documentation files are not necessary); for the writing tasks, you should submit your answer of each task; for each checkpoint and each challenge task, you should attach images, videos, or typescript files, along with a brief explanation.

3.1 Overall Instructions

- **Start EARLY!** It is likely this lab is the most time-consuming project (e.g. more than the sum of *shell lab*, *malloc lab* and *proxy lab* of ICS). It is your responsibility, rather than the TAs', to schedule the work.

- As a developer of system software, you need to design your program architecture carefully and wisely. You may find the following tasks difficult to get started unless you are an experienced developer of "big" projects. Don't get panic. Take efforts, and you will be capable of finishing them eventually.

- We won't cover every detail of the protocol stack; only the specified interfaces must be implemented. Feel free to cover the unspecified details as your wish, as long as you comply with the protocols.

- Feel free to make use of any tool and skill to program and debug; `vnetUtils` and `Wireshark` are recommended but not restricted to. Besides, you also need to write some debug code yourself for every task.

3.2 Link-layer: Packet I/O on Ethernet

For many reasons, Link-layer is quite complex. One of them is that you can translate bit stream into frames in many different ways. In this part, we will work with Ethernet II frames. Ethernet II is supported by Linux virtual ethernet device and almost all GigE NICs. In other words, you can use Ethernet II frames to communicate with your wired router/switch, or other nodes in a virtual network on your Linux machine.

Programming Task 1 (PT1). Use `libpcap` to implement following methods to support network device management. Note: The method signatures in this writeup are only for references. Feel free to design your version.

```
/**
 * @file device.h
 * @brief Library supporting network device management.
 */

/**
 * Add a device to the library for sending/receiving packets.
 *
 * @param device Name of network device to send/receive packet on.
 * @return A non-negative _device-ID_ on success, -1 on error.
 */
int addDevice(const char* device);

/**
 * Find a device added by `addDevice`.
 *
 * @param device Name of the network device.
 * @return A non-negative _device-ID_ on success, -1 if no such device
 * was found.
 */
int findDevice(const char* device);
```

Programming Task 2 (PT2). Use `libpcap` to implement the following methods to support sending/receiving Ethernet II frames.

```
/**
 * @file packetio.h
 * @brief Library supporting sending/receiving Ethernet II frames.
 */

#include <netinet/ether.h>

/**
 * @brief Encapsulate some data into an Ethernet II frame and send it.
 *
 * @param buf Pointer to the payload.
 * @param len Length of the payload.
 * @param ethtype EtherType field value of this frame.
 * @param destmac MAC address of the destination.
 * @param id ID of the device(returned by `addDevice`) to send on.
 * @return 0 on success, -1 on error.
 * @see addDevice
 */
int sendFrame(const void* buf, int len,
int ethtype, const void* destmac, int id);

/**
 * @brief Process a frame upon receiving it.
 *
 * @param buf Pointer to the frame.
 * @param len Length of the frame.
 * @param id ID of the device (returned by `addDevice`) receiving
 *        current frame.
 * @return 0 on success, -1 on error.
 * @see addDevice
 */
typedef int (*frameReceiveCallback)(const void*, int, int);

/**
 * @brief Register a callback function to be called each time an
 *        Ethernet II frame was received.
 *
 * @param callback the callback function.
 * @return 0 on success, -1 on error.
 * @see frameReceiveCallback
 */
int setFrameReceiveCallback(frameReceiveCallback callback);
```

Checkpoint 1 (CP1). Show that your implementation can detect network interfaces on the host.

Checkpoint 2 (CP2). Show that your implementation can capture frames from a device and inject frames to a device using libpcap.

3.2.1 Hints and Instructions

- Check your byte order!
- In this part, you don't need to compute the CRC bits when building an ethernet frame.

3.3 Network-layer: IP Protocol

After completing part 1 you should be able to send something to some other place via Ethernet (if there're no fragmented packets!), but as you've learnt, the range is strictly limited. In this part, you will implement a simplified version of IP protocol, version 4 to expand this range to the whole Internet.

Programming Task 3 (PT3). Update the method in `device.h`, Use the library in `packetio.h` to implement the following methods to support sending/receiving IP packets.

You should follow RFC791 when working on this.

```
/**
 * @file ip.h
 * @brief Library supporting sending/receiving IP packets encapsulated
     in an Ethernet II frame.
 */

#include <netinet/ip.h>

/**
 * @brief Send an IP packet to specified host.
 *
 * @param src Source IP address.
 * @param dest Destination IP address.
 * @param proto Value of ‘protocol‘ field in IP header.
 * @param buf pointer to IP payload
 * @param len Length of IP payload
 * @return 0 on success, -1 on error.
 */
int sendIPPacket(const struct in_addr src, const struct in_addr dest,
    int proto, const void *buf, int len);

/**
 * @brief Process an IP packet upon receiving it.
 *
 * @param buf Pointer to the packet.
 * @param len Length of the packet.
 * @return 0 on success, -1 on error.
 * @see addDevice
 */
typedef int (*IPPacketReceiveCallback)(const void* buf, int len);

/**
 * @brief Register a callback function to be called each time an IP
    packet was received.
 *
 * @param callback The callback function.
 * @return 0 on success, -1 on error.
 * @see IPPacketReceiveCallback
 */
int setIPPacketReceiveCallback(IPPacketReceiveCallback callback);

/**
 * @brief Manully add an item to routing table. Useful when talking
    with real Linux machines.
 *
 * @param dest The destination IP prefix.
 * @param mask The subnet mask of the destination IP prefix.
 * @param nextHopMAC MAC address of the next hop.
 * @param device Name of device to send packets on.
 * @return 0 on success, -1 on error
 */
```

```
int setRoutingTable(const struct in_addr dest,
    const struct in_addr mask,
    const void* nextHopMAC, const char *device);
```

Writing Task 1 (WT1). `sendFrame()` requires the caller to provide the destination MAC address when sending IP packets, but users of IP layer will not provide the address to you. Explain how you addressed this problem when implementing IP protocol.

Writing Task 2 (WT2). Describe your routing algorithm.

Checkpoint 3 (CP3). Use tcpdump/wireshark to capture the IP packets generated by your implementation. Hexdump the content of *any one* packet here, and show meanings for each byte (for example, "the first byte is 0x50 and the most significant 4 bits of the first byte is 0101, it means ...; and ...").

Checkpoint 4 (CP4). *Use vnetUtils or other tools* to create a virtual network with the following topology and show that: (1) ns1 can discover ns4; (2) after we disconnect ns2 from the network, ns1 cannot discover ns4; (3) after we connect ns2 to the network again, ns1 can discover ns4.

```
ns1 --- ns2 --- ns3 --- ns4
```

Checkpoint 5 (CP5). Create a virtual network with the following topology and show the distances between each pair of hosts. The distance depends on your routing algorithm. After that, disconnect ns5 from the network and show the distances again.

```
ns1 --- ns2 --- ns3 --- ns4
         |       |
        ns5 --- ns6
```

Checkpoint 6 (CP6). Show the "longest prefix matching" rule applies in your implementation.

3.3.1 Hints and Instructions

- Check your byte order!!
- When designing a usable system, you should always take all corner cases into consideration. Describe what corner cases your system met and your solution during the development process in the writing tasks.
- You are not required to implement IP packet fragmentation, simply drop fragmented packets is OK.
- You are not required to support TOS and IP options. Use a default value when sending, ignore them when receiving.
- Your implementation should be reentrant.

3.4 Transport-layer: TCP Protocol

With the help of IP protocol, you can talk with any host in the network now. In this part, you will implement a simplified version of TCP protocol, providing a subset of POSIX-compatible socket interfaces to the applications.

Programming Task 4 (PT4). Use the interfaces provided by `ip.h` to implement the following POSIX-compatible interfaces.

You should follow RFC793 when working on this. You are also expected to keep compatibility with POSIX.1-2017 standard when implementing your socket interfaces, but that's just for your applications to run correctly, you will **NOT** lose credits if your interfaces behave slightly different.

Note: You can use a file descriptor allocating algorithm slightly different from the standardized one to avoid conflicts with fds allocated by the system.

```
/**
 * @file socket.h
 * @brief POSIX-compatible socket library supporting TCP protocol on
    IPv4.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

/**
 * @see [POSIX.1-2017:socket](http://pubs.opengroup.org/onlinepubs/
 * 9699919799/functions/socket.html)
 */
int __wrap_socket(int domain, int type, int protocol);

/**
```

```
 * @see [POSIX.1-2017:bind](http://pubs.opengroup.org/onlinepubs/
 * 9699919799/functions/bind.html)
 */
 int __wrap_bind(int socket, const struct sockaddr *address,
    socklen_t address_len);

/**
 * @see [POSIX.1-2017:listen](http://pubs.opengroup.org/onlinepubs/
 * 9699919799/functions/listen.html)
 */
int __wrap_listen(int socket, int backlog);

/**
 * @see [POSIX.1-2017:connect](http://pubs.opengroup.org/onlinepubs/
 * 9699919799/functions/connect.html)
 */
int __wrap_connect(int socket, const struct sockaddr *address,
    socklen_t address_len);

/**
 * @see [POSIX.1-2017:accept](http://pubs.opengroup.org/onlinepubs/
 * 9699919799/functions/accept.html)
 */
int __wrap_accept(int socket, struct sockaddr *address,
    socklen_t *address_len);

/**
 * @see [POSIX.1-2017:read](http://pubs.opengroup.org/onlinepubs/
 * 9699919799/functions/read.html)
 */
ssize_t __wrap_read(int fildes, void *buf, size_t nbyte);

/**
 * @see [POSIX.1-2017:write](http://pubs.opengroup.org/onlinepubs/
 * 9699919799/functions/write.html)
 */
ssize_t __wrap_write(int fildes, const void *buf, size_t nbyte);

/**
 * @see [POSIX.1-2017:close](http://pubs.opengroup.org/onlinepubs/
 * 9699919799/functions/close.html)
 */
int __wrap_close(int fildes);

/**
 * @see [POSIX.1-2017:getaddrinfo](http://pubs.opengroup.org/
    onlinepubs/
 * 9699919799/functions/getaddrinfo.html)
 */
int __wrap_getaddrinfo(const char *node, const char *service,
    const struct addrinfo *hints,
    struct addrinfo **res);
```

Writing Task 3 (WT3). Describe how you correctly handled TCP state changes.

Checkpoint 7 (CP7). Use tcpdump/wireshark to capture the TCP packets generated by your implementation. Hexdump the content of any one packet here, and show meanings for each byte in the TCP header.

Checkpoint 8 (CP8). Show your implementaion provides reliable delivery (i.e., it can detect packet loss and retransmit the lost packets). You are encouraged to attach a screenshot of the wireshark packet trace here. Check section 4.2 to see how to emulate a lossy link.

Checkpoint 9 (CP9). Create a virtual network with the following topology and run `echo_server` at ns4 and `echo_client` at ns1. The source code is under the folder called "checkpoints". Paste the output of them here. Note that you are not allowed to make changes to the source code (i.e., the `*.h` and `*.c` files). Check out section 4 to see how to hijack the library functions such as `listen()`.

```
ns1 --- ns2 --- ns3 --- ns4
```

Checkpoint 10 (CP10). Create a virtual network with the following topology and run `perf_server` at ns4 and `perf_client` at ns1. Paste the output of them here. Again, you are not allowed to make changes to the source code.

```
ns1 --- ns2 --- ns3 --- ns4
```

3.4.1 Hints and Instructions

- Check your byte order!!!
- Carefully describe the corner cases your system can meet and your solution in the writing tasks.
- You are not required to support TCP options other than those specified in RFC793.
- You are not required to support any socket option.
- You are not required to support TCP out-of-band data.
- You are not required to support TCP flow control.

- You are not required to support TCP congestion control.
- You are not required to implement any sort of URL resolving mechanism. When implementing `getaddrinfo`, you only need to let it work when:
  - `node` is a valid IPv4 address or `NULL`
  - `service` is a valid port number or `NULL`
  - `hints` has `.ai_family == AF_INET`, `.ai_socktype == IPPROTO_TCP`, `.ai_flags == 0` or `hints == NULL`
- Your implementation should be reentrant, as specified by POSIX standard.
- Although TCP flow control is not required, you still need to set the `window` field in TCP header properly when sending and try not to make bytes-in-flight value larger than the receive window to let your TCP work correctly.
- To be robust, your implementation of `read`, `write` and `close` should fall back to the real library functions when processing fds not allocated by yourself.
- Being POSIX-compatible doesn't necessarily require you to implement all functionalities implemented by Linux. Just check the arguments and fail if required feature is not to be implemented.

3.5 Bonus: Test/Evaluation

This is an open task. In this part, you will come up with a way to prove that your program works robustly and efficiently by yourself and do evaluation, just like what you will do for your works to be submitted :) There's only one constraint: when using applications to test your TCP implementation, your application must use standard POSIX socket interfaces. In other words, the applications should run correctly on Linux machines.

Programming Task 5 (PT5). Implement a method to initialize your evaluating environment.

Writing Task 4 (WT4). Describe your evaluating methodology, result and usage of your evaluation method.

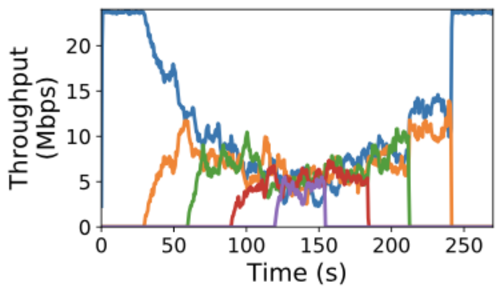

Figure 1: Five flows depart one-by-one on a 24 Mbit/s link.

3.6 Challenge

This part is optional.

Challenge 1 (CL1). Show that your implementation handles correctly simultaneous connection synchronization (i.e., two users call `connect()` to initiate connection simultaneously) and simultaneous connection close (i.e., two users call `close()` to close simultaneously).

Challenge 2 (CL2). Provide fragmentation and reassembly of long IP datagrams at Layer 3.

Challenge 3 (CL3). Provide weighted fair queuing at routers. Evaluate your implementation with at least 2 flows with different weights.

Challenge 4 (CL4). Provide TCP flow control. Test your implementation with a fast sender and a slow receiver.

Challenge 5 (CL5). Provide TCP congestion control. Show how congestion window (*cwnd*) size changes over time.

Challenge 6 (CL6). Implement the BBR congestion control. The kernel implementation may be a good reference: `https://github.com/torvalds/linux/blob/master/net/ipv4/tcp_bbr.c`. Note: If you completed CL6, the CL5 would also be done.

Challenge 7 (CL7). Show that your congestion control algorithm provides fairness when 5 TCP flows share a bottleneck. Figure 1 is an example.

Challenge 8 (CL8). Write an HTTP proxy built atop your protocol stack. Test your proxy with a browser such as Chrome or Firefox.

Challenge 9 (CL9). Write at least 10 unit tests for your implementation. We recommend Google Testing framework for unit testing.

Challenge 10 (CL10). Provide coverage test and generate test report. We recommend Gcov for coverage test and its front-end Lcov for generating the report in HTML format.

Challenge 11 (CL11). Other interesting topics are welcome.

4 Hints for Testing (Playing with) Your Program

As mentioned above, we won't provide a standardized way for you to test/evaluate your implementation, it's your responsibility to complete this part by yourself. But unfortunately, this requires quite a handful of knowledge that shouldn't be required in a network lab focusing on TCP/IP. This section describes some useful techniques to empower you to force network programs to use your TCP interface, testing your program in an emulated network, and communicate with real Linux machines. However, you are encouraged to explore how those and more things work and get your hands on them.

4.1 Hijack Library Functions

We won't be happy if we must develop some brand-new example program using our TCP interface just to test it. So here comes the problem: If a network program uses functionalities already implemented by us only, can we force it to use our implementations?

Yes, one approach is using an alternate library of functions when compiling a program from source:

By specifying `--wrap [fun]` when invoking `ld`, any undefined reference to `[fun]` will be resolved to `__wrap_[fun]`, any undefined reference to `__real_[fun]` will be resolved to `[fun]`. By specifying `-Wl` option when invoking `gcc`, you can pass options to the linker.

You may use some open-source programs, such as iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool, to do the evaluation. But these programs may use some tough functions (e.g. `fork()`) which is not required to support in this lab, so you can choose whether to use such tools or to implement one by yourself.

Example: to hijack `socket()`, implement a function named `__wrap_socket` with desired functionality, then specify `-Wl,--wrap,socket` will let the program use your version of `socket()`, you can still call the real `socket()` by calling `__real_socket()`.

Note: You may want to implement `send/recv`, `sendto/recvfrom` as a wrapper of `write/read` to support programs using those functions.

4.2 Emulate *Bad* Links

It's hard to test if your program can survive specific corner case on a typical local/wired (virtual) link because packet losses and congestions are very unlikely to happen. To emulate such a link, you can use `tc-netem` provided by Linux itself. You can thus add delay, loss to a link, or limit its bandwidth.

You can find more examples at networking:netem [Linux Foundation Wiki].

4.3 Disable the Kernel Protocol Stack

By using `libpcap`, we can monitor the packet flow and send packets. But the kernel protocol stack won't be disabled, which is not desired by us. `vnetUtils` can help us disable the kernel stack.

If you are not using `vnetUtils`, you can also disable the IP module by filtering packets using `iptables`. To disable the IP module on specified host, use `filter` table to drop all packets:

```
# Drop all packets
iptables -t filter -I FORWARD -j DROP
iptables -t filter -I INPUT -j DROP
iptables -t filter -I OUTPUT -j DROP
# Revert
iptables -t filter -D FORWARD -j DROP
iptables -t filter -D INPUT -j DROP
iptables -t filter -D OUTPUT -j DROP
```

4.4 Talk with Real Linux Machines

If you comply with the protocols carefully, the hosts running your protocol stack should be able to communicate with real machines running an off-the-shelf operating system with network protocol stack. To talk with a Linux machine (or machines running Windows, BSD, etc), you need to configure the routing table on all the hosts on the path correctly. Implementing an automatic (and often distributed) solution using packets is hard, but things can be easier with some manual methods. The problem comes in two directions: for you, you need to know how to reach the peer. This can be easily done by calling `setRoutingTable()` implemented by yourself earlier to set routing table manually. Similarly, for others, they need to know how to reach you. This can be done using `ip route`.

```
# Set routing table
ip route add [your IP/subnet] via [next hop IP] dev [device name]
# Revert
ip route del [your IP/subnet] via [next hop IP] dev [device name]
```

Specially, if the host is in your LAN, you should use the script below instead:

```
# Set routing table
ip route add [your IP/subnet] dev [device name]
# Revert
ip route del [your IP/subnet] dev [device name]
```

Note: Carefully check your implementation before talking to a real machine. Linux network protocol stack is filled with sanity checks, the kernel may drop a strange-looked packet without notifying anybody. Besides, a real machine may include other options not specified in this lab, e.g. offloading checksum in NIC hardware, so please pay double attention.

Can your protocol stack talk with other students' designs?

5 Grading

This lab worths 100 points (pts). The table below shows the pts for each problem.

	pt	Total pt
PT1, PT2	Each task 5 pts	10
PT3, PT4	Each task 8 pts	16
CP1-CP8	Each checkpoint 5 pts	40
CP9, CP10	Each checkpoint 7 pts	14
WT1		4
WT2, WT3	Each task 8 pts	16
PT5, WT4	Extra points for bonus!	+20
CL1-CL11	Each task 8 pts. No more than 30 pts.	+30

6 Project Submission

In this lab, you should submit a directory named `lab2` containing the following items in an archive named `lab2-[your name]-[your student ID].[tar|tar.gz|zip]`:

- `src/`

  Source code of your programs, including the protocol stack and your self-built programs for evaluating.

- `Makefile` or `CMakeLists.txt`

  Makefile for building your program.

- `README.pdf`

  A report of your lab (in pdf format *only*), including your solutions to writing tasks, your explanations for checkpoints, and your code arrangement (especially for each programming task; a codelist is enough).

- `not-implemented.pdf`

  A single document (in pdf format *only*) listing the features that are specified by this document/the standard but you didn't implement. You also need to explicitly give the reason why you are not implementing it for each feature. Submit an empty file if you completely finished all the tasks.

  You may lose credits by adding items to this list, but being dishonest on this will make you lose more!

- `checkpoints/`

  Images, videos, or typescript files for each checkpoints.

For each submission, please send an email with title `lab2_Name_StudentID` to chengke@pku.edu.cn. Missing the deadlines incurs a penalty.